MilkyWay@home Python Package Documentation (*mwahpy* v1.4.0)

Tom Donlon

Department of Physics, Applied Physics and Astronomy
Rensselaer Polytechnic Institute

Contents

1 Introduction

1.1 What is MilkyWay@home?

At best, "MilkyWay@home" is a nebulous term that can refer to several different projects, softwares, and/or groups of people. At its core, MilkyWay@home is a crowd-sourced supercomputer that takes volunteer computing time to perform complex and time-consuming calculations that are designed to improve our understanding of the Milky Way Galaxy. At the moment, this includes both the MilkyWay@home Separation application and the MilkyWay@home N-body application. The MilkyWay@home project was developed at Rensselaer Polytechnic Institute under the direction of Dr. Heidi Jo Newberg, and has utilized countless hours of work from many graduate students and researchers. For more information regarding MilkyWay@home, I direct the user to the MilkyWay forums at `https://milkyway.cs.rpi.edu/milkyway/`. All of the MilkyWay@home code is publicly available at `https://github.com/Milkyway-at-home`.

While the Separation application is an important part of MilkyWay@home, the Separation application is difficult to apply to other projects. Additionally, the tools that I have developed for the Separation application are quite situation specific and not designed with user friendliness/adaptability in mind. For this reason, *mwahpy* does not include functionality for the Separation application. For Separation application tools and documentation, see the Separation application github page at `https://github.com/Milkyway-at-home/milkywayathome_client/tree/master/separation`.

What I refer to as "MilkyWay@home" in this document is actually the MilkyWay@home N-body software. This software was originally designed to generate model dwarf galaxies in the Milky Way gravitational potential and integrate them forwards in time. This evolved dwarf galaxy would then be compared to observations of stellar streams in the sky in order to determine a quality of fit. By optimizing over the parameters that were used to generate the dwarf galaxy, one could in theory determine the parameters of the progenitor dwarf galaxy. This could even be done for streams with no obvious progenitor (e.g. the Orphan Stream).

It was realized shortly thereafter that the N-body application could be used to integrate the orbits of any bodies in the Milky Way, not just those that were generated as part of a dwarf galaxy. The ability to insert list of bodies to integrate forwards in time was provided. Additionally, the ability to control the size of the timestep of the simulation, the orbit of a dwarf galaxy progenitor, the timescale of the simulation, the dwarf galaxy parameters, the underlying gravitational potential, and other MilkyWay@home functionality made it a versatile and robust tool for N-body integration.

While the MilkyWay@home N-body application is currently used by many graduate and undergraduate students at Rensselaer Polytechnic Institute, the MilkyWay@home team is making an effort to encourage more widespread usage of the MilkyWay@home N-body software. Our goal is to provide a user-friendly, comprehensive experience for N-body integration. This will decrease the time needed to get new students and researchers up to speed with N-body software, which has become commonplace in the dynamical astrophysics community.

We expect that the MilkyWay@home N-body application will become more widely used, and that proper documentation and accompanying tools will quickly become necessary. Additionally, new applications of the N-body software have expanded simulations for upwards of one million bodies. Analysis of these large simulations requires quick, robust software. It is for these reasons that *mwahpy* was developed.

1.2 What is *mwahpy*?

The MilkyWay@home N-body software has been used by the Galactic dynamics research group at Rensselaer Polytechnic Institute for years. As each new student comes into contact with the software, each one has had to develop their own tools for analyzing the outputs of the software. Each individual software has its own set of bugs, sets the student back a few weeks while developing their own tools, and by not using a standardized set of analysis software we open up the group to problems with sharing data and code. Additionally, MilkyWay@home is written in **C**, a language which can be daunting to the typical undergraduate student (and is often unecessarily unwieldly if you are just trying to cut data or make a plot). My goal to alleviate these issues was a compilation of the tools that I had built in python and tested over my tenure as an undergraduate (and improved during my time as a graduate student) in the MilkyWay@home dynamics group. This became the *mwahpy* package.

Before we talk about what *mwahpy* is, it is good to clarify what *mwahpy* is not. This package is not able to do N-body integration. Things like generating dwarf galaxies, running simulations, and performing routines on the MilkyWay@home supercomputer are to be delegated to the proper MilkyWay@home software. Additionally, I would like to express that because the Separation application is interacted with by many fewer people than the N-body application, *mwahpy* will not support Separation application functionality.

There is plenty that *mwahpy* is meant to be used for, though. At its very base, *mwahpy* is a python package that is designed to easily and quickly read in N-body data ouput from the MilkyWay@home N-body software. Once read in, it is fairly simple to cut the data, plot the data, and save the data in a variety of formats. This package is also able to output data in a format that is readable by the MilkyWay@home N-body software. The software will automatically calculate complicated values such as proper motion and energies as needed, instead of computing everything up front. The major benefit of these routines is that new users can be confident that the values that are produced by *mwahpy* have been tested over several years and are known to be accurate.

There are several associated auxiliary packages in *mwahpy* as well, such as the coordinate transformation subpackage and the orbit fitting subpackages. These are provided in the package because while working with N-body simulations, I often ran into situations where I used the functionality provided in the auxiliary subpackages. The auxiliary subpackages are less streamlined and complete compared to the main subpackages, but are still fairly well tested and can be trusted.

The code for *mwahpy* is publically available and can be found at `https://github.com/thomasdonlon/mwahpy`. Not only is collaboration on the code allowed, it is encouraged! If you would like functionality added to *mwahpy* or if you find any bugs in the code, you can either leave a comment on the github page or you can write the code yourself and make a pull request. This code is still being actively maintained, and I plan on eventually getting around to any bug fixes or desired functionality that are brought to my attention.

1.3 What can *mwahpy* do?

The following functionality is provided in *mwahpy*:

- Easily & quickly read in data from a MilkyWay@home N-body *.out* file
- MilkyWay@home N-body output only provides 3D Cartesian positions and velocities, as well as mass (in MilkyWay@home structural units). The *mwahpy* package converts this data into many other useful forms, such as R.A. & Dec., Galactic longitude and latitude, angular momenta about the Galactic center, line-of-sight velocity, proper motions, and others.
- Make cuts based on any of *mwahpy*'s supported values, cut the data based on individual components of the data, or subsample the data randomly or symmetrically.
- Quickly plot N-body data in any of the supported values
- A variety of coordinate transformations for typical coordinate systems used in Galactic astronomy
- Two separate orbit fitting routines that have been used in publications

The proper syntax, usage, and details of each of the above capabilities will be outlined in the following sections.

1.4 Citing *mwahpy*

If *mwahpy* or any of its related code is used in a publication or project, I simply ask that you state that *mwahpy* was used in your work and provide the link for the *mwahpy* github repository located at `https://github.com/thomasdonlon/mwahpy`. There is currently no publication that should be cited for this package. In the future, the requirements for proper citation may change, so please check again before final publication. Thank you for using *mwahpy*!

2 Installing *mwahpy*

Installing *mwahpy* is as simple as installing any other python package. For linux machines, go to your terminal and type

Code Block 1: Installing *mwahpy*

```
>>> pip3 install mwahpy
```

Alternatively, you can also install *mwahpy* using

Code Block 2: Installing *mwahpy*

```
>>> python3 -m pip install mwahpy
```

It may be necessary to install *mwahpy* only in your user directory due to user access restrictions. **It is strongly recommended you do not use sudo to install python packages.** Using **sudo** makes it difficult to track who has the permissions to use what package on your machine, and on machines with multiple users, it can mean you have to install it for each user individually anyways. Misuse of **sudo** can also result in accidentally using a different version of *mwahpy* than intended, particularly after updates. To install *mwahpy* for only the active user, you can instead type

Code Block 3: Installing *mwahpy*

```
>>> pip3 install mwahpy --user
```

or

Code Block 4: Installing *mwahpy*

```
>>> python3 -m pip install mwahpy --user
```

Any of these lines of code will install *mwahpy* on your machine, as well as install all of the prerequisite packages needed for *mwahpy* to work.

If you use Anaconda as a package manager (as is becoming more and more common these days), it is recommended that you still use `pip` to install *mwahpy* as is shown above. Hoewever, this can have a couple disadvantages: For starters, the Conda package manager will not manage *mwahpy*. However, *mwahpy* will still be managed by the Anaconda environment. Additionally, you can run into some problems if your default installation of python is different than your Anaconda installation of python (a more common problem than it should be). **If your default python installation is different than your Anaconda installation, I strongly suggest that you go through and fix your python configuration, as you will almost certainly run into other problems eventually.** However, if you don't want to do that (or have your python installation purposefully configured that way for some reason), you can use

Code Block 5: Installing *mwahpy*

```
>>> conda skeleton pypi mwahpy
>>> conda build mwahpy
>>> conda install --use-local mwahpy
```

This will install *mwahpy* and avoid issues with a misconfigured Anaconda/python installation. Additionally, if *mwahpy* is installed this way, the Conda package manager will be responsible for managing *mwahpy* instead of `pip`. **Be aware that this method may not automatically update *mwahpy* when a new vrsion of the package is released.**

In order to use *mwahpy* after it is installed, simply import the subpackages as you would any other other package in your python script. For example, the following lines are all syntactically acceptable imports:

Code Block 6: Importing *mwahpy*

```
import mwahpy.output_handler
from mwahpy.timestep import Timestep
from mwahpy.coords import *
```

It should be noted that *mwahpy* is only built for and maintained for python v3.2.3 and above. You should get an error when installing if your python installation is not recent enough for *mwahpy* . Depending on your installation, you may instead have to type `>>> python -m pip install mwahpy` instead of `python3`, or use `pip3` instead of `pip`. If you are experiencing errors, asking questions on the *mwahpy* github page or trying different permutations of these installation methods are recommended.

If you wish to install *mwahpy* for development, then you should clone the most recent *mwahpy* github repository (found at `https://github.com/thomasdonlon/mwahpy`). Feel free to make your own fork of the master branch if that's what you would prefer. Then, you should download the source code wherever you wish on your machine. You will then need to build the *mwahpy* package on your machine so that you can test any changes you make to the code. Navigate to the directory where *mwahpy* is stored and then type in a terminal:

Code Block 7: Using *mwahpy* as a developer

```
>>> pip3 uninstall mwahpy
>>> python3 setup.py develop
```

Whenever you make a change to the *mwahpy* source code, you will have to run the `python3 setup.py develop` line again in a terminal to rebuild the package on your machine. Any changes that you feel are beneficial to the package should be sent as a pull request to the master branch. When you are done making changes, you should return to your terminal and run

Code Block 8: Using *mwahpy* as a developer

```
>>> python3 setup.py develop --uninstall
>>> pip3 install mwahpy
```

This will return your system to a configuration where it is using the most recent stable release of the *mwahpy* package. If you intend on running code using the changes you have made to *mwahpy*, you will have to run that code before returning your *mwahpy* build to the current PyPi version.

3 Core Functionality

WARNING:

MilkyWay@home uses a right-handed Galactic Cartesian coordinate system. This is defined as positive X being in the direction of the Sun towards the Galactic center, positive Y being in the direction of the disk spin at the location of the Sun, and positive Z being in the direction of the right-handed cross product X×Y (often called the "Galactic north"). In our coordinate system, the Sun is located at the position (X,Y,Z) = (-8,0,0).

This is in contrast to many other Galactic scientists, who prefer a left-handed coordinate system where the X-axis is flipped and the Sun is located at (X,Y,Z) = (8,0,0). Most notably, the *galpy* package is left-handed. In a left-handed coordinate system the physical interpretation of certain quantities are not always clear (such as the right-handed angular momentum cross product). Many coordinate transformations in *mwahpy* allow for left-handed coordinates. However, be aware that by default, *mwahpy* (and MilkyWay@home) output is right-handed.

3.1 The *Timestep* Class

The `Timestep` class is the heart of *mwahpy*, and it's where the majority of the important and useful calculations in the package are performed. An instance of the `Timestep` class represents a single timestep of an N-body simulation. In other words, a `Timestep` instance is the data from a single MilkyWay@home `.out` file. The code for the `Timestep` class can be found in *mwahpy*'s `timestep.py` file.

3.1.1 Initializing a *Timestep*

The most basic usage of `Timestep` object is a blank instance, to which you can then manually add data:

Code Block 9: Blank Timestep

```
import numpy as np
from mwahpy.timestep import Timestep

t = Timestep()

t.typ = np.array([0, 0, 1])
t.id = np.array([0, 1, 2])
t.x = np.array([10, 50, 100])
t.y = np.array([12, 15, 17])
t.z = np.array([1, 2, 3])
t.vx = np.array([13, 183, 102])
t.vy = np.array([0, 50, 180])
t.vz = np.array([23, 69, 12])
t.mass = np.array([1, 1, 1])
```

These 9 values are all that you need to specify for the `Timestep` class to do its job. These 9 values (typ, id, x, y, z, vx, vy, vz, and mass) will be referred to as *"provided values"*. This is in contrast to the "*supported values*", which are any values that *mwahpy* can calculate for you (a full list of the supported values is provided in Section 6.9). I will also often refer to supported values as "calculated" values. In the `Timestep` implementation, all of the particle IDs, x positions, etc. are stored in order as a `numpy` array of those values. As such, the *mwahpy* package heavily utilizes features of the `numpy` package, and some knowledge of `numpy` can be helpful for those working with *mwahpy*.

If you wish to print the information of a single particle, you can do so with `print_particle()` by specifying the ID of the particle you are interested in:

Code Block 10: Printing data for a single particle

```
>>> t.printParticle(1)
Printing data for Particle 1:
(typ: 0, id:1, x:50, y:15, z:2, vx:183, vy:50, vz:1, mass:1, )
```

WARNING:

The provided values should **always** be `numpy` arrays of identical length. Attempting to calculate values, plot data, or write out data when the provided values have mismatched length or are not `numpy` arrays will typically result in an error. It is strongly recommended that the user uses the built-in `Timestep` methods for cutting or adding data instead of doing it manually.

From this point forward, the user can ask for any *mwahpy* supported value, and it will be calculated for them. For example, if you wanted the line-of-sight velocities of the particles, you could type:

Code Block 11: Calculating supported values

```
>>> t.vlos
array([  7.894394  , 235.8965911 , 128.90974589])
```

Note that if you call `t.print_particle()` again (this time using the optional `dec` argument to shorten the output), it now shows more data for the particle!

Code Block 12: Printing data for a single particle

```
>>> t.print_particle(1, dec=2)
Printing data for Particle 1:
(typ:0, id:1, x:50, y:15, z:2, vx:183, vy:50, vz:69, mass:1, msol:222288.47,
l:0.29, b:0.04, ra:266.54, dec:-28.67, dist:59.94, lx:935, ly:3084,
lz:-245, lperp:3222.62, ltot:3231.92, r:52.24, R:52.2, vlos:235.9,
vgsr:247.14, rad:192.15, rot:-4.69, distFromCOM:52.24, )
```

This was all calculated in the background when you asked the package to calculate `vlos` for this `Timestep`. Due to the overhead on some calculations, the more computationally complex calculations are avoided until the user requests those values.

The `Timestep` class has one very unique property: attributes of a `Timestep` instance can be accessed via the usual method, or as the key to a dictionary. In fact, comparing the two methods shows that these two actions produce equivalent results.

Code Block 13: Accessing Timestep attributes

```
>>> t.x #accessing data as an attribute
array([ 10,  50, 100])
>>> t['x'] #accessing data as a dict key
array([ 10,  50, 100])
>>> np.all(t.x == t['x'])
True
>>> t.x[0] = 1 #changing the value for Particle 0's x position
>>> t.x[0] #the same value is accessed by both methods
1
>>> t['x'][0]
1
>>> t.x[0] == t['x'][0]
True
```

At first this property may seem confusing and not particularly useful. What is the point of being able to access the same data in two different ways? It turns out that adding this functionality to `Timestep` allows for some rather powerful behavior. Of the greatest importance to the typical user is the implementation of the iterator for `Timestep`, which iterates over the *keys* of the class. In this case, that has been implemented as the names of all of the supported values (that have been calculated so far!).

Code Block 14: Iterating over a Timestep

```
>>> outstr = ''
>>> for key in t:
...     outstr += (key + ', ')
>>> print(outstr)
id, x, y, z, vx, vy, vz, mass, msol, l, b, ra, dec, dist, lx, ly,
lz, lperp, ltot, r, R, vlos, vgsr, rad, rot, distFromCOM,
>>> outstr = ''
>>> for key in t:
...     outstr += (str(round(t[key][0],2)) + ', ')
>>> print(outstr)
0, 10, 12, 1, 13, 0, 23, 1, 222288.47, 0.88, 0.06, 266.86, -28.15, 21.66,
276, 217, -156, 351.09, 384.19, 15.65, 15.62, 7.89, 21.43, 9.77, -9.99,
15.65,
```

This may not seem like a big deal at first. However, if one tries to reproduce this behavior for a class that doesn't have the property `c.x == c['x']`, then you will quickly run into several problems. I expect that the typical reader would try to iterate over the class' built-in attribute dictionary, which is what I initially tried. What happens if you want to add attributes to the class that cannot be iterated over at the same time as your arrays, such as a single identifying string `c.name`? I suggest that the reader play around with this idea on their own if they are so inclined.

3.1.2 Reading In & Writing Out Data

So far, we know how to initialize a `Timestep`, how to add data to it manually, and how to access this data. This is all fine and good, but most of the time a user will be interested in using data that has already been generated by the MilkyWay@hoe N-body application. Conveniently, there is a *mwahpy* function built specifically for this in the `mwahpy.output_handler` subpackage:

Code Block 15: Reading in a Timestep

```
>>> import mwahpy.output_handler as oh
>>> t = oh.read_output('<path/to/mwahpy>/test/test.out')
Reading in data from ../test/test.out...
[-------------->     ] 73%
10 objects read in
Converting data...done
>>> t.x
array([ 4.53813066, -0.59395581, -1.41000385, -0.00613003,  0.11393187,
        0.38487162,  0.35803147,  0.3565866 ,  0.52268562, 3.57179938])
```

The `oh.read_output` function takes in the path to a MilkyWay@home `.out` file and outputs a `Timestep` instance of the data from that file. In the above code block, we read in the test `,out` file, which is provided in `.../mwahpy/test/`. The `oh.read_output` function provides you with a progress bar, which is useful for large files. In this case, the progress bar finished almost instantly since the file was small. Typically, the progress bar will reach 100%. The `oh.read_output` function also provides you with the number of particles that were read in from the file, and a brief update on when it is converting the data (again, useful for very large files). From this point forward, you can perform any normal `Timestep` operations on the new `Timestep` object.

The `oh.read_output` function has been heavily optimized, and operates extremely quickly. Even for files with millions of values, reading in the data only takes a few seconds on most machines. If you are working with N-body output, it is strongly recommended that you use this function to initialize your `Timestep` instances.

The *mwahpy* package also offers a few different options for printing out `Timestep` data to a file. Notably, *mwahpy* can print out the data to a `.csv` file,

Code Block 16: Writing out a Timestep

```
>>> oh.make_csv(t, '/path/to/my/file.csv')
Writing header...
Printing data...
Timestep output to /path/to/my/file.csv
```

Saving the data to a `.csv` saves all of the supported values that has been calculated so far. Alternatively, you can write out data from a `Timestep` to a MilkyWay@home `.in` file,

Code Block 17: Writing out a Timestep

```
>>> oh.make_nbody_input(t, '/path/to/my/file.in')
Writing Timestep as N-body input to /path/to/my/file.in...
done
```

Note that a MilkyWay@home `.in` file will only include the data about the 8 provided values, and not any of the other supported values. This is due to the format that MilkyWay@home requires readable files to be in. Any file that you generate with `make_nbody_input` is immediately ready to be used as the manual body input for a MilkyWay@home N-body simulation.

WARNING:

The MilkyWay@home N-body client has a bug that will cause crashes if it tries to read in a `.in` file that includes any baryonic (`typ == 0`) particles. If you plan on using a `.in` file as a manual body file for the N-body client, you should run

```
t[typ] = np.array([1]*len(t))
```

before writing that timestep out to a `.in` file. This will not change the physics of the simulation, although it is annoying for tracking certain types of particles.

If a user wishes to suspend a `Timestep` object for later use, I recommend the `pickle` package (`https://docs.python.org/3/library/pickle.html`). The `pickle` package allows for object serialization and saving out/reading in arbitrary objects as binary code.

3.1.3 Manipulating Data

Aftet N-body data has been read in, it is useful to be able to only select the data that you are interested in. There are a few different routines for this, namely `subset_rect()` and `subset_circ()`. These methods are n-dimensional cutting routines. For example, say that you took the test data and only wanted particles with a positive X value.

Code Block 18: Cutting data in a Timestep

```
>>> tcopy = t.copy()
>>> tcopy.subset_rect(['x'], [(0,1000)]) #[axes], [(lower lim, upper lim)]
>>> tcopy.x
array([4.53813066, 0.11393187, 0.38487162, 0.35803147, 0.3565866 ,
       0.52268562, 3.57179938])
>>> tcopy.y
array([-0.05701174,  0.39540971, -0.43899208,  0.14577695,  0.0082146 ,
       -0.34261058,  0.53908404])
```

Note that `subset_rect()` has cut along **all** axes, not only the axis that we instructed it to cut along. As such, it is easy to see that `tcopy.y` has the same length as `tcopy.x`. It is also important to note that we made a copy of `t` before using `subset_rect()`, because the `subset` methods treat the `Timestep` object as directly mutable.

While cutting the data, we provided `1000` as an upper value for X. This was because `subset_rect()` requires both a lower and upper limit for each axis when cutting the data. To get around this, we just chose some arbitrarily large value that would not cut any data off at the high end of X values.

You can also cut on multiple axes with `subset_rect()`. Say that you wanted to only include particles with positive X and negative Y values:

Code Block 19: Cutting data in a Timestep

```
>>> tcopy = t.copy()
>>> tcopy.subset_rect(['x','y'], [(0,1000),(-1000,0)])
>>> tcopy.x
array([4.53813066, 0.38487162, 0.52268562])
>>> tcopy.y
array([-0.05701174, -0.43899208, -0.34261058])
```

As all methods in `Timestep`, these routines can be performed on any supported values **after they have been calculated**. Attempting to cut on line-of-sight velocity before it has been calculated, for example, would result in a `KeyError`.

Similar results can be obtained from `subset_circ()`. If we wanted all particles with X values within 1 kpc of the Galactic center:

Code Block 20: Cutting data in a Timestep

```
>>> tcopy = t.copy()
>>> tcopy.subset_circ(['x'],[1],[0])
>>> tcopy.x
array([-0.59395581, -0.00613003,  0.11393187,  0.38487162,  0.35803147,
        0.3565866 ,  0.52268562])
```

Or if we wanted all particles with X and Y values within 1 kpc of the Galactic center:

Code Block 21: Cutting data in a Timestep

```
>>> tcopy = t.copy()
>>> tcopy.subset_circ(['x', 'y'],[1,1],[0,0])
>>> tcopy.x
array([-0.00613003,  0.11393187,  0.38487162,  0.35803147,  0.3565866 ,
        0.52268562])
>>> tcopy.y
array([ 0.43348783,  0.39540971, -0.43899208,  0.14577695,  0.0082146 ,
       -0.34261058])
```

This routine is a bit of a misnomer, as it can be extended to any n-dimensional spheroidal volume of your value space. For example, say that we wanted particles located within a spheroid defined with an semiaxis of length 1 on the X-axis, a semiaxis of length 2 on the Y-axis, and a semiaxis of length 5 on the Z-axis, centered at (X, Y, Z) = (0, 1, 3). This can be done with

Code Block 22: Cutting data in a Timestep

```
>>> tcopy = t.copy()
>>> tcopy.subset_circ(['x', 'y', 'z'],[1,2,5],[0,1,3])
>>> tcopy.x
array([-0.59395581, -0.00613003,  0.11393187,  0.35803147,  0.3565866 ])
>>> tcopy.y
array([0.87634781, 0.43348783, 0.39540971, 0.14577695, 0.0082146 ])
>>> tcopy.z
array([ 0.55739028, -0.18463063,  0.32116868,  0.17315727, -0.09600029])
```

Unlike `subset_rect()`, this routine becomes somewhat unclear when you begin mixing values with different units. In that case, it is suggested that you either make multiple circular cuts where the units of each value match, or make rectangular cuts instead.

Other data manipulation routines include sampling and splitting `Timesteps`. There are three types of sampling that *mwahpy* supports: manual, incremental, and random sampling, shown below.

Code Block 23: Sampling a Timestep

```
>>> #manual sampling
>>> tcopy = t.copy()
>>> tcopy.take([0,2,3,6,9]) #take particles at indices 0, 2, 3, 6, and 9
>>> tcopy.x
array([ 4.53813066, -1.41000385, -0.00613003,  0.35803147,  3.57179938])

>>> #incremental sampling
>>> tcopy = t.copy()
>>> tcopy.subsample(2) #take every 2nd particle, starting at 0
>>> tcopy.x
array([ 4.53813066, -1.41000385,  0.11393187,  0.35803147,  0.52268562])

>>> #random sampling
>>> tcopy = t.copy()
>>> tcopy.rand_sample(5) #randomly take 5 particles from the data
>>> tcopy.x
array([ 4.53813066, -0.59395581,  0.38487162, -0.00613003,  0.35803147])
>>> #run again to demonstrate randomness
>>> tcopy = t.copy()
>>> tcopy.rand_sample(5)
>>> tcopy.x
array([ 0.52268562, -0.59395581,  0.3565866 , -0.00613003,  0.11393187])
```

It is easy to split a `Timestep` by providing the index at which you want to split:

Code Block 24: Splitting a Timestep

```
>>> tcopy, tcopy2 = t.split(4) #split at index 4
>>> tcopy.x
array([ 4.53813066, -0.59395581, -1.41000385, -0.00613003])
>>> tcopy2.x
array([ 0.11393187,  0.38487162,  0.35803147,  0.3565866,   0.52268562,
3.57179938])
```

Finally, if you would like to add two `Timestep` instances together, use the `append_timestep()` or `append_point()` methods:

Code Block 25: Adding Timesteps together

```
>>> tcopy.append_timestep(tcopy2) #append two Timesteps
>>> tcopy.x
array([ 4.53813066, -0.59395581, -1.41000385, -0.00613003,  0.11393187,
        0.38487162,  0.35803147,  0.3565866 ,  0.52268562,  3.57179938])

>>> tcopy, tcopy2 = t.split(4)
>>> tcopy.append_point(tcopy2, n=5) #add tcopy2 index 5 to tcopy
>>> tcopy.x
array([ 4.53813066, -0.59395581, -1.41000385, -0.00613003,  3.57179938])
```

Note that actions like just replacing a single `x` value in a `Timestep` does not update the rest of the calculated values in the `Timestep`. That `Timestep` will need to be updated manually using `Timestep.update()`. This method will update all values associated with the `Timestep` in order to keep them in sync with the provided values.

WARNING:

Note that if the `auto_update` flag is turned on (it is on by default), *mwahpy* functions will automatically keep the provided and calculated values in sync. However, if you manually update the provided values after the calculated values have been computed, you will need to run `Timestep.update()`.

Code Block 26: Updating a Timestep

```
>>> t = oh.readOutput('../test/test.out')

Reading in data from ../test/test.out...
[-------------->     ] 73%
10 objects read in
Converting data...done
>>> t.x
array([ 4.53813066, -0.59395581, -1.41000385, -0.00613003,  0.11393187,
        0.38487162,  0.35803147,  0.3565866 ,  0.52268562,  3.57179938])
>>> t.dist_from_com
Calculating basic values...
array([4.53771154, 1.20665236, 2.01267674, 0.48791978, 0.51046082,
       0.99322694, 0.39540509, 0.34974823, 0.85795412, 3.73818636])
>>> t.x[0] = 0
>>> t.dist_from_com
array([4.53771154, 1.20665236, 2.01267674, 0.48791978, 0.51046082,
       0.99322694, 0.39540509, 0.34974823, 0.85795412, 3.73818636])
>>> t.update()
>>> t.dist_from_com
array([0.80324921, 1.29094239, 2.05506687, 0.36725208, 0.51176956,
       0.99371407, 0.34057982, 0.29000981, 0.81102414, 3.37390626])
```

You should now have a fundamental undertsanding of how to use the `Timestep` class. The rest of the methods that were not covered in this section can be found in Section 6.9. Continue reading for information about handling entire simulations with multiple timesteps and plotting timesteps and simulations.

3.2 The *Nbody* Class

The `Nbody` class is one rung above the `Timestep` class in the *mwahpy* hierarchy; one `Nbody` instance is composed of many individual `Timestep` instances. This is useful when you need to compare many timesteps from a single MilkyWay@home simulation. The simplest example of a `Nbody` instance is the default instance:

Code Block 27: Initializing an Nbody

```
>>> from mwahpy.nbody import Nbody
>>> n = Nbody()
```

From here, you can add a `Timestep` to the dictionary in the `Nbody` instance:

Code Block 28: Initializing an Nbody

```
>>> import mwahpy.output_handler as oh
>>> t = oh.read_output('<path/to/mwahpy>/test/test.out')
Reading in data from ../test/test.out...
[-------------->     ] 73%
10 objects read in
Converting data...done
>>> n[1] = t
>>> n[1].x
array([ 4.53813066, -0.59395581, -1.41000385, -0.00613003,  0.11393187,
        0.38487162,  0.35803147,  0.3565866 ,  0.52268562,  3.57179938])
```

Note that the `Nbody` instance itself does not store the `Timestep` data, but just points to it.

Code Block 29: An Nbody does not store simulation data

```
>>> n.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Nbody' object has no attribute 'x'
```

In this case, the `Timestep` `t` still exists on its own, but the `Nbody` instance forms a nice functional container for it. This can be seen by altering `t`, as the change is also present when accessing the data through the `Nbody`. Thus, the `Nbody` structure does not copy the `Timestep` data, it just references it.

Code Block 30: An Nbody points to Timestep data

```
>>> t.x[0] = 0
>>> t.x
array([ 0.        , -0.59395581, -1.41000385, -0.00613003,  0.11393187,
        0.38487162,  0.35803147,  0.3565866 ,  0.52268562,  3.57179938])
>>> n[1].x
array([ 0.        , -0.59395581, -1.41000385, -0.00613003,  0.11393187,
        0.38487162,  0.35803147,  0.3565866 ,  0.52268562,  3.57179938])
```

The `Timestep` does recognize that it belongs to an `Nbody` class however, and it remembers its designated time and the `Nbody` object which it belongs to:

Code Block 31: Accessing an Nbody

```
>>> t.time
1
>>> t.nbody
<mwahpy.nbody.Nbody object at 0x[...]>
```

WARNING:

Currently, `Timestep` instances only weakly retain information about parent `Nbody` instances. If a `Timestep` is added to a second `Nbody`, it will replace its `self.time` and `self.nbody` from the first `Nbody` instance with the new information from the second `Nbody`. This is made more confusing by the fact that multiple `Nbody` instances can point to the same `Timestep`, or a single `Nbody` can point to the same `Timestep` more than once. In these cases, the data can be properly accessed through the `Nbody` objects, but is not guaranteed to refer back to the desired information when accessed from the `Timestep` object.

3.2.1 Reading In & Writing Out Data

As with the `Timestep` class, the `Nbody` class is typically not created manually, but is initialized by reading in data. This is also done through the `mwahpy.output_handler` subpackage by providing a folder with the `oh.read_folder()` function.

Code Block 32: Reading data into an Nbody

```
>>> n = oh.read_folder('<...path/to/mwahpy>/mwahpy/test/nbody_test/')
Reading in data from directory ../test/nbody_test/...

Reading in data from ../test/nbody_test/1...
[-------------->     ] 73%
10 objects read in
Converting data...done

Reading in data from ../test/nbody_test/2...
[-------------->     ] 73%
10 objects read in
Converting data...done

Reading in data from ../test/nbody_test/3...
[-------------->     ] 73%
10 objects read in
Converting data...done
>>> n[1].x
array([ 4.53813066, -0.59395581, -1.41000385, -0.00613003,  0.11393187,
        0.38487162,  0.35803147,  0.3565866 ,  0.52268562,  3.57179938])
>>> n[2].x
array([21.51019042, 17.97537383, 15.99136357, 12.30622142, 10.80896895,
       10.56226125, 12.94243578, 11.16187291, 11.17781449,  8.04042707])
```

This data has been read in from the `../mwahpy/test/nbody_test/` folder that is included in *mwahpy* .

WARNING:

When reading in a `Nbody` from simulation data, the **only** files in the provided folder must be MilkyWay@home `.out` files, and **must** be named only their time **without the** `.out` **extension**. This is the default output scheme of MilkyWay@home N-body.

Currently, there is not support for writing out `Nbody` data. This can be done fairly simply by iterating over the `Nbody` (see the next subsection for details) and writing out each `Timestep`. It is also not recommended that a user writes out `Nbody` data using the `pickle` package.

3.2.2 Functionality of *Nbody* Objects

`Nbody` objects have a few nice capabilities that allow for robust treatment of simulation data. To begin, iterating over an `Nbody` object returns its component `Timestep` objects sorted by time.

Code Block 33: Iterating over an Nbody

```
>>> for ts in n:
...     print(ts.time)
1
2
3
```

Being able to read in and iterate over large groups of `Timestep` instances from the same simulation is much easier than going through each one individually, not to mention it is often faster.

The times of each timestep can be scaled in order to reflect the actual physical simulation time of each timestep using the `scale_times()` method. For example, if one timestep is equal to 0.5 Myr in physical time, then we can scale the corresponding timesteps. We see that the original information is retained, although renamed:

Code Block 34: Scaling Nbody times

```
>>> n.scale_times(0.5)
>>> for ts in n:
...     print(ts.time)
0.5
1.0
1.5
>>> n[0.5].x
array([ 4.53813066, -0.59395581, -1.41000385, -0.00613003,  0.11393187,
        0.38487162,  0.35803147,  0.3565866 ,  0.52268562,  3.57179938])
```

3.3 Plotting

This section will be added at a later date. The plotting capabilities of *mwahpy* are not very developed at this time. I suggest using `pyplot` if you want to make plots of MilkyWay@home data.

4 Auxiliary Subpackages

The *mwahpy* package comes with a few auxiliary subpackages with additional content that the typical user might find useful. The `mwahpy.orbit_fitter` and the `mwahpy.orbit_fitter_gc` subpackages have been used in academic contexts, and are refactored for general use. The `mwahpy.coords` subpackage contains a number of useful coordinate transforms that one might find handy when working with MilkyWay@home N-body data.

4.1 Coordinate Transformations

The coordinate transformation package `mwahpy.coords` is made up of many first and second-order transformations for position, velocity, and other values. A complete list of the routines and functions provided in this package is given in Section 6.1.

This subpackage contains most of the coordinate transformations that a typical person will need to do Galactic astronomy, excluding some common transformations (such as R.A., Dec. to l, b) that are implemented in the `astropy` python package, among others.

4.2 Orbit Fitting

At many times in studies of Galactic dynamics, given a collection of points in phase space, one may want to fit an orbit to them. Luckily, *mwahpy* comes with two different orbit fitting routines to do just that. The first, `mwahpy.orbit_fitter`, fits an orbit in a heliocentric frame, while the second, `mwahpy.orbit_fitter_gc`, fits an orbit in a "naturalized" Galactocentric frame. They both have their uses: for many simple orbits `mwahpy.orbit_fitter` will achieve a better goodness-of-fit in a shorter amount of time compared to `mwahpy.orbit_fitter_gc`, but for complicated orbits the opposite is often true.

The math behind these routines was used in Donlon et al. (2019) to fit orbits to structures, and is discussed therein. For more information, I also refer the reader to Willett et al. (2009).

Remember – orbit fitting is as much of an art as it is a science. Tweaking your "errors", timesteps, evolve times, and other parameters may all be necessary for you to obtain good orbit fits. Additionally, since differential evolution is a non-deterministic algorithm, it may be desirable to run several orbit fits of the same observed data in order to get a variety of fit orbits that you can choose your favorite from. This will also provide a good idea of the limitations of the orbit fitting algorithm and the errors in its fits for your data.

4.2.1 *orbit_fitter*

Given some observed data for a group of stars, one can calculate a naive orbit by simply taking the average of their positions and velocities. However, if these stars are at substantially different positions in their orbits (say, for example, in a stellar stream) then the velcoities of these stars also will vary as a function of position along the stream, and the average velocity is no longer expected to give a good estimate for the orbit of the stream at any given position.

In order to fit an orbit to this collection of stars, one can imagine generating a model orbit with your best guess of parameters. This orbit will differ from the actual positions and velocities of the star data by some measurable amount. In *mwahpy* we use a goodness-of-fit (GoF) adapted from Willett et al. (2009) to compute the difference between the model orbit and the observed data,

$$GoF = \sum_{i=1}^{n} \sum_{j=1}^{m} \left(\frac{(\theta_{i,j,model} - \theta_{i,j,obs})}{\sigma_{i,j,obs}} \right)^2,$$

where i spans the n number of data dimensions in the observed data $(l, b, d, v_x, v_y, v_z, v_{GSR})$, j spans the m number of datapoints (stars) observed, $\theta_{i,j}$ corresponds to the ith dimension of the jth observed datapoint, and $\sigma_{i,j}$ corresponds to the error measured in $\theta_{i,j}$. For example, $\theta_{d,7,model}$ corresponds to the heliocentric distance data in the model at the location of the observed 7th star. This approach generalizes the GoF so that the orbit fitting routine can still measure a GoF even if some or all of the velocity dimensions of the observed data are unavailable.

This GoF is then minimized using the `scipy.optimize.differential_evolution()` routine. Without going too deep into the math behind this algorithm, here are the basics: a population of possible orbits is generated, and their GoFs are measured. New "child" possible orbits are generated by randomly crossing the current population of possible orbits, and then their GoFs are evaluated. If any of the children's GoFs are improvements over their parents, then those improved child possible orbits are inserted into the population. The process is repeated until either a tolerance bound is reached or the maximum number of iterations has been achieved. The best orbit fit is the possible orbit model with the lowest GoF value at the end of this process, which is then reported to the user as the optimized orbit.

This algorithm uses Galactic longitude as its orbital position parameter, which is why it does not ask for error in l, which is assumed to be zero. If l does not work well as your orbital parameter (for example, if your stars are all located in a tidal stream that is perpendicular to the Galactic plane from our line of sight, or is oriented radially away or towards our line of sight) then the `orbit_fitter_gc` routine will probably do a better job fitting your orbit.

The GoF also includes a cost based on orbital position and timestep resolution. Each degree separating an orbit in galactic longitude and a data point adds 1000 to the GoF. This is to prevent orbit fits that do not actually span the data, or that pass the data with poor timestep resolution.

Below is an example of an orbit fit using points generated from an actual orbit:

Code Block 35: Fitting a Mock Orbit

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mwahpy.orbit_fitter as of

>>> #parameters for the mock orbit
>>> test_o_l = 90
>>> test_o_b = 20
>>> test_o_d = 25
>>> test_o_vx = 150
>>> test_o_vy = -200
>>> test_o_vz = 100

>>> #sample the mock orbit
>>> ts = np.linspace(0, 0.25, 1000)*u.Gyr
>>> sample = np.array([100, 250, 400, 500, 600, 750, 850])
>>> o = of.make_orbit([test_o_l, test_o_b, test_o_d, test_o_vx, test_o_vy,
 test_o_vz])

>>> l = np.take(o.ll(ts), sample)
>>> b = np.take(o.bb(ts), sample)
>>> d = np.take(o.dist(ts), sample)

>>> plt.scatter(l, b, s=5) #plot the mock sky position data
>>> plt.scatter(o.ll(ts), o.bb(ts), s=1)
>>> plt.show()

>>> plt.scatter(l, d, s=5) #plot the mock position/dist data
>>> plt.scatter(o.ll(ts), o.dist(ts), s=1)
>>> plt.show()

>>> print('Goodness-of-Fit␣of␣actual␣values:')
>>> print(chi_squared([test_o_l, test_o_b, test_o_d, test_o_vx, test_o_vy,
 test_o_vz], data=test_orbit_data))

>>> params, x2 = of.fit_orbit(l, b, b_err, d, d_err) #the orbit fitting

>>> of.plot_orbit_gal(l, b, d, params) #plots of the orbit fit
>>> of.plot_orbit_icrs(l, b, d, params)
```

If you aren't interested in typing all this code in and running it yourself, that's fine. It's implemented as `mwahpy.orbit_fitter.test()`, and computes a pretty good orbit fit in roughly 5 min on my machines. Depending on the number of cores available on your machine, this may take longer or shorter for you.

4.2.2 *orbit_fitter_gc*

Theoretically, all orbiting structures should lie along an orbital plane that passes through the Galactic center. This means that if one were to look out at something like a tidal stream from the Galactic center, that tidal stream would approximately trace out a great circle on the sky.

This orbit fitter takes advantage of that fact by performing the same process as `mwahpy.orbit_fitter`, except instead of using Galactic longitude as the orbital position parameter (and therefore a heliocentric frame), this routine attempts to compute the orbital plane of the data, and then uses the longitudinal angle

within that plane as its orbital parameter. This means that we are fitting the orbits in a Galactocentric frame, which is a more "natural" choice of coordinates for structures such as tidal streams.

The plane fitting is done via ordinary least squares (OLS, see the appendix of Newby et al. 2013); however, unlike in that work, we constrain our plane to pass through the Galactic center. This is done by finding an approximate solution to the system

$$\mathbf{A}\vec{n} = \vec{b},$$

where, in a slight abuse of notation, $\mathbf{A} = [\ \vec{x}, \vec{y}\ ]^T$, and $\vec{b} = \vec{z}^T$. This approximate solution is found via the pseudo-inverse of $\mathbf{A}$,

$$\vec{n} \approx \left(\left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}\right)\vec{b} = \langle\ A,\ \ B,\ \ C\ \rangle.$$

This solution minimizes the sum of the squares of distances between the data and a plane with equation

$$A\hat{x} + B\hat{y} + C\hat{z} = 0,$$

.

The OLS plane has a unit normal vector $\hat{n}$,

$$\hat{n} = \frac{\langle\ A,\ \ B,\ \ C\ \rangle}{||\vec{n}||}$$

which determines the direction of the new Z' axis. We also choose an orthogonal "point" unit vector that is constrained to lie in the $X - Z$ plane, $\hat{p}$. This vector is defined as

$$\hat{p} = \langle\ \left((\hat{n}_x/\hat{n}_z) + 1\right)^{-1/2},\ \ 0,\ \ -\hat{p}_x\hat{n}_x/\hat{n}_z\ \rangle,$$

and corresponds to the direction of the new X' axis. Finally we define a third unit vector $\hat{o} = \hat{n} \times \hat{p}$, which corresponds to the direction of the Y'-axis. Then, the data is rotated into a new coordinate frame where the X'-Y' plane lies coincident with the OLS plane, and the Z'-axis is aligned along the normal vector to the OLS plane. This can be done easily through a change-of-basis matrix, which is how it is implemented in *mwahpy* :

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \hat{p}_x & \hat{p}_y & \hat{p}_z \\ \hat{o}_x & \hat{o}_y & \hat{o}_z \\ \hat{n}_x & \hat{n}_y & \hat{n}_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}.$$

Then, the new longitude and latitude for each star are defined as

$$\Lambda = \arctan(y'/x')$$

$$\beta = \arcsin(z'/r),$$

where r is the spherical Galactocentric radius of that star.

Then, the same procedure as in the previous section is used to optimize an orbit to the data, except the values of (Λ, β) are used instead of (l, b). Similar to the above section, `mwahpy.orbit_fitter_gc` also has a `test()` routine that you can use to ensure that the orbit fitter works.

5 Flags & Settings

5.1 *flags*

The flags and settings for *mwahpy* can be found in `flags.py`. Currently, there are only a few options, but more may be added in the future.

- `mwahpy.flags.auto_update` (default value = 1)

  This flag determines whether or not to automatically update an entire `Timestep` whenever the provided values for that `Timestep` change **due to a *mwahpy* method or function**. Having this flag turned on will keep things like the center of mass and momentum accurate even after changing the constituent data, and will keep all of the calculated values in sync with the provided values throughout *mwahpy* functions.

  If you are frequently updating certain values in a `Timestep`, this flag can occasionally cause performance issues (if this flag is causing performance issues, that's probably a sign that there's a better way to do what you are trying to do). It is strongly recommended that this flag is only turned off if the user understands what updating means, how to update the data manually, and what data needs to be updated.

- `mwahpy.flags.progress_bars` (default value = 1)

  If this flag is turned on, *mwahpy* will use progress bars when performing certain actions (such as reading in files) in order to avoid long wait times without updating the terminal. If turned off, *mwahpy* functionality will not be changed; however, be aware that you may occasionally experience long periods of time where your terminal is silent.

- `mwahpy.flags.verbose` (default value = 1)

  This flag dictates how much is printed out into the terminal when *mwahpy* code is running. If turned on, functions are much noisier, and will update you on their current activity. If this is turned off, the overall functionality of *mwahpy* will remain unchanged, but the terminal will be less informative.

5.2 *mwahpy_glob*

Constants and functions that are universally accessed in *mwahpy* are stored in `mwahpy_glob.py`.

- `mwahpy.mwahpy_glob.file_len(f)`

  Given a file, this function will determine the number of lines in that file. Useful for implementing `mwahpy.mwahpy_glob.progress_bar()` while reading in files.

  Parameters:

  - `f` (str) : The filename (should be a path).

  Returns:

  - len (int) : The number of lines in the file.

- `mwahpy.mwahpy_glob.G`

  Newton's gravitational constant is provided in `astropy` units of $m^3/kg\ s^2$.

- `mwahpy.mwahpy_glob.kms_to_kpcgyr`

  The conversion factor from km/s to kpc/Gyr. Roughly equal to unity (1.023).

- `mwahpy.mwahpy_glob.kpcgyr_to_kms`

  The conversion factor from kpc/Gyr to km/s. Roughly equal to unity (0.978).

- `mwahpy.mwahpy_glob.progress_bar(value, endvalue, bar_length=20)`

  Adapted from `https://stackoverflow.com/questions/6169217/replace-console-output-in-python`. This function can be placed inside a loop and will output a nice progress bar in the terminal, provided you know the end value that the loop should terminate on.

  Parameters:

  - `value` (int) : The value that is being iterated. Does not need to be increased by 1 each time – can be any amount.
  - `endvalue` (int) : The value at which the loop terminates .
  - `bar_length` (int, optional) : How many characters long the progress bar is in the terminal output.

  Returns:

- `mwahpy.mwahpy_glob.struct_to_sol`

  This is the conversion factor between MilkyWay@home structural masses and solar masses. Equal to 222,288.47 solar masses/structure unit.

5.3 *pot*

Constants and functions related to the Milky Way's gravitational potential are found in `mwahpy.pot.py`. The default potential that is used in *mwahpy* is the Orphan Stream Fit #5 Potential from Newberg et al. (2010). This potential is very similar to Law et al. (2005).

- `mwahpy.pot.energy_offset`

  Due to the logarithmic halo potential term in our gravitational potential for the Milky Way, the potential does not disappear as one travels infinitely far from the center of the potential. A side effect of this type of halo potential is that it is not clear what structures are bound or unbound to the potential, as the zero-point of the energy is not representative of the escape velocity. This adjusts the calculated potential energy so that it is consistent with Donlon et al. (2019), and so that clearly bound structures will not have positive energy values. The default value is -60,000 km^2/s^2. Changing this value does not change the physics of the system, it just changes the energy of a particle at infinity.

- `mwahpy.pot.m_bulge`

  The mass of the bulge in solar mass `astropy` units. Equal to 3.4×10^{10} $M_\odot$

- `mwahpy.pot.m_disk`

  The mass of the disk in solar mass `astropy` units. Equal to 1.0×10^{11} $M_\odot$

- `mwahpy.pot.v_halo`

  The Milky Way halo dispersion in `astropy` km/s. Equal to 74.61 km/s.

- `mwahpy.pot.plot_potential(potential, Rrange=[0.01,10.])`

  This method will generate and show a figure with the rotation curve of the potential in the plane of the Milky Way.

  Parameters:

  - `potential` (`galpy potential` object) : the potential that you wish to graph
  - `Rrange` (list of floats) : the Galactocentric cylindrical radius of the points at which you want to evaluate the potential

  Returns:

- `mwahpy.pot.pot`

  The total gravitational potential of the Milky Way. Given as a `galpy potential` object.

- `mwahpy.pot.pot_bulge`

  The gravitational potential of the Milky Way due to the bulge. Given as a `galpy potential` object.

- `mwahpy.pot.pot_disk`

  The gravitational potential of the Milky Way due to the disk. Given as a `galpy potential` object.

- `mwahpy.pot.pot_halo`

  The gravitational potential of the Milky Way due to the halo. Given as a `galpy potential` object.

6 Functions & Methods

Below is a complete list of functions and methods implemented in the files of *mwahpy* that perform actual routines. Note that only the functions and methods that are meant for the typical end user are provided in detail here, and therefore some helper functions/methods that are present in the code are not gone over in this document. For a tutorial of using the main parts of *mwahpy*, one should look at Section 3.

6.1 *coords*

- `mwahpy.coords.cart_to_cyl(x, y, z)`

  Takes in Cartesian coordinates and returns cylindrical coordinates. Supports array-like inputs.

  Parameters:

  - `x` (float or array-like floats) : The Cartesian X coordinate(s) of the data.
  - `y` (float or array-like floats) : The Cartesian Y coordinate(s) of the data.
  - `z` (float or array-like floats) : The Cartesian Z coordinate(s) of the data.

  Returns:

  - `R` (float or array-like floats) : The cylindrical radius coordinate(s) of the data.
  - `z` (float or array-like floats) : The Cartesian Z coordinate(s) of the data.
  - `phi` (float or array-like floats) : The azimuthal angle(s) of the data, in degrees.

- `mwahpy.coords.cart_to_gal(x, y, z, left_handed=False)`

  Takes in Galactocentric Cartesian coordinates and returns Galactic coordinates. Supports array-like inputs. Uses a right-handed system by default.

  Parameters:

  - `x` (float or array-like floats) : The Galactocentric Cartesian X coordinate(s) of the data.
  - `y` (float or array-like floats) : The Galactocentric Cartesian Y coordinate(s) of the data.
  - `z` (float or array-like floats) : The Galactocentric Cartesian Z coordinate(s) of the data.
  - `left_handed` (bool, optional) : If `True`, a left-handed Galactocentric Cartesian system is used.

  Returns:

  - `l` (float or array-like floats) : The Galactic longitude coordinate(s) of the data, in degrees.
  - `b` (float or array-like floats) : The Galactic latitude coordinate(s) of the data, in degrees.
  - `r` (float or array-like floats) : The heliocentric distance(s) of the data in whatever units of distance the inputs were in.

- `mwahpy.coords.cart_to_lambet(x, y, z, normal, point)`

  Takes in Galactocentric Cartesian coordinates, the normal vector to a plane, and a point to suggest the new X-axis and returns longitude and latitude coordinates with the X-Y plane orthogonal to the normal vector and the Z-axis along the normal vector. The longitude (Lambda) increases counter-clockwise as viewed looking on the plane from along the positive Z-axis, and the latitude (Beta) increases in the direction of the normal vector and is zero in the plane.

  Parameters:

  - `x` (float or array-like floats) : The Galactocentric Cartesian X coordinate(s) of the data.
  - `y` (float or array-like floats) : The Galactocentric Cartesian Y coordinate(s) of the data.
  - `z` (float or array-like floats) : The Galactocentric Cartesian Z coordinate(s) of the data.

- `normal` (array-like of floats) : The three-vector Cartesian coordinates for the normal vector of the plane.
- `point` (array-like of floats) : The three-vector Cartesian coordinates for the vector that suggests the new X-axis of the plane. Does not need to be orthogonal to the normal vector.

Returns:

- `Lam` (float or array-like floats) : The longitude coordinates of the points in the new planar coordinates.
- `Bet` (float or array-like floats) : The latitude coordinates of the points in the new planar coordinates.

- `mwahpy.coords.cart_to_lonlat(x, y, z)`

  Takes in arbitrary Cartesian coordinates and returns longitude and latitude coordinates for that Cartesian coordinate system. The longitude (Lambda) increases counter-clockwise as viewed looking downwards on the plane from along the positive Z-axis, and the latitude (Beta) increases in the direction of the Z-axis and is zero in the plane.

  Parameters:

  - `x` (float or array-like floats) : The Cartesian X coordinate(s) of the data.
  - `y` (float or array-like floats) : The Cartesian Y coordinate(s) of the data.
  - `z` (float or array-like floats) : The Cartesian Z coordinate(s) of the data.

  Returns:

  - `Lam` (float or array-like floats) : The longitude coordinates of the points.
  - `Bet` (float or array-like floats) : The latitude coordinates of the points.

- `mwahpy.coords.cart_to_plane(x, y, z, normal, point)`

  Takes in Galactocentric Cartesian coordinates, the normal vector to a plane, and a point to suggest the new X-axis and returns shifted Cartesian coordinates with the X-Y plane orthogonal to the normal vector and the Z-axis along the normal vector.

  Parameters:

  - `x` (float or array-like floats) : The Galactocentric Cartesian X coordinate(s) of the data.
  - `y` (float or array-like floats) : The Galactocentric Cartesian Y coordinate(s) of the data.
  - `z` (float or array-like floats) : The Galactocentric Cartesian Z coordinate(s) of the data.
  - `normal` (array-like of floats) : The three-vector Cartesian coordinates for the normal vector of the plane.
  - `point` (array-like of floats) : The three-vector Cartesian coordinates for the vector that suggests the new X-axis of the plane. Does not need to be orthogonal to the normal vector.

  Returns:

  - `x` (float or array-like floats) : The Cartesian X coordinates of the points in the new planar coordinates.
  - `y` (float or array-like floats) : The Cartesian Y coordinates of the points in the new planar coordinates.
  - `z` (float or array-like floats) : The Cartesian Z coordinates of the points in the new planar coordinates.

- `mwahpy.coords.cart_to_sgr(x, y, z)`

  Takes in Galactocentric Cartesian coordinates and returns Sagittarius Tidal Stream longitude and latitude coordinates. See Majewski et al. (2003) for more info about the coordinate system.

  Parameters:

  - `x` (float or array-like floats) : The Galactocentric Cartesian X coordinate(s) of the data.
  - `y` (float or array-like floats) : The Galactocentric Cartesian Y coordinate(s) of the data.
  - `z` (float or array-like floats) : The Galactocentric Cartesian Z coordinate(s) of the data.

  Returns:

  - `Lam` (float or array-like floats) : The longitude coordinates of the points in Sgr coordinates.
  - `Bet` (float or array-like floats) : The latitude coordinates of the points in Sgr coordinates.

- `mwahpy.coords.cart_to_sph(x, y, z)`

  Takes in Cartesian coordinates and returns spherical coordinates. Supports array-like inputs.

  Parameters:

  - `x` (float or array-like floats) : The Cartesian X coordinate(s) of the data.
  - `y` (float or array-like floats) : The Cartesian Y coordinate(s) of the data.
  - `z` (float or array-like floats) : The Cartesian Z coordinate(s) of the data.

  Returns:

  - `phi` (float or array-like floats) : The azimuthal angle(s) of the data, in degrees.
  - `theta` (float or array-like floats) : The polar angle(s) of the data, in degrees.
  - `r` (float or array-like floats) : The spherical radius coordinate(s) of the data.

- `mwahpy.coords.cyl_to_cart(R, z, phi)`

  Takes in cylindrical coordinates and returns Cartesian coordinates. Supports array-like inputs.

  Parameters:

  - `R` (float or array-like floats) : The cylindrical radius coordinate(s) of the data.
  - `z` (float or array-like floats) : The Cartesian Z coordinate(s) of the data.
  - `phi` (float or array-like floats) : The azimuthal angle(s) of the data, in degrees.

  Returns:

  - `x` (float or array-like floats) : The Cartesian X coordinate(s) of the data.
  - `y` (float or array-like floats) : The Cartesian Y coordinate(s) of the data.
  - `z` (float or array-like floats) : The Cartesian Z coordinate(s) of the data.

- `mwahpy.coords.cyl_to_gal(R, z, phi)`

  Takes in Galactocentric cylindrical coordinates and returns Galactic coordinates. Supports array-like inputs.

  Parameters:

  - `R` (float or array-like floats) : The cylindrical radius coordinate(s) of the data.
  - `z` (float or array-like floats) : The Cartesian Z coordinate(s) of the data.
  - `phi` (float or array-like floats) : The azimuthal angle(s) of the data, in degrees.

  Returns:

- `l` (float or array-like floats) : The Galactic longitude coordinate(s) of the data, in degrees.
- `b` (float or array-like floats) : The Galactic latitude coordinate(s) of the data, in degrees.
- `r` (float or array-like floats) : The heliocentric distance(s) of the data in whatever units of distance the inputs were in.

- `mwahpy.coords.gal_to_cart(l, b, r, left_handed=False, rad=False)`

  Takes in Galactic coordinates and returns Galactocentric Cartesian coordinates. Supports array-like inputs. Uses a right-handed system by default.

  Parameters:

  - `l` (float or array-like floats) : The Galactic longitude coordinate(s) of the data, in degrees by default.
  - `b` (float or array-like floats) : The Galactic latitude coordinate(s) of the data, in degrees by default.
  - `r` (float or array-like floats) : The heliocentric distance(s) of the data.
  - `left_handed` (bool, optional) : If `True`, a left-handed Galactocentric Cartesian system is used.
  - `rad` (bool, optional) : If `True`, input is given in radians. Otherwise, input should be in degrees.

  Returns:

  - `x` (float or array-like floats) : The Galactocentric Cartesian X coordinate(s) of the data. In whatever units of distance the input was in.
  - `y` (float or array-like floats) : The Galactocentric Cartesian Y coordinate(s) of the data. In whatever units of distance the input was in.
  - `z` (float or array-like floats) : The Galactocentric Cartesian Z coordinate(s) of the data. In whatever units of distance the input was in.

- `mwahpy.coords.gal_to_cyl(l, b, r)`

  Takes in Galactic coordinates and returns Galactocentric cylindrical coordinates. Supports array-like inputs.

  Parameters:

  - `l` (float or array-like floats) : The Galactic longitude coordinate(s) of the data, in degrees.
  - `b` (float or array-like floats) : The Galactic latitude coordinate(s) of the data, in degrees.
  - `r` (float or array-like floats) : The heliocentric distance(s) of the data.

  Returns:

  - `R` (float or array-like floats) : The cylindrical radius coordinate(s) of the data, in whatever units the input distance was in.
  - `z` (float or array-like floats) : The Cartesian Z coordinate(s) of the data, in whatever unit the input distance was in.
  - `phi` (float or array-like floats) : The azimuthal angle(s) of the data, in degrees.

- `mwahpy.coords.gal_to_lambet(l, b, r, normal, point)`

  Takes in Galactic coordinates, the normal vector to a plane, and a point to suggest the new X-axis and returns longitude and latitude coordinates with the X-Y plane orthogonal to the normal vector and the Z-axis along the normal vector. The longitude (Lambda) increases counter-clockwise as viewed looking on the plane from along the positive Z-axis, and the latitude (Beta) increases in the direction of the normal vector and is zero in the plane. WARNING: This is heliocentric.

  Parameters:

  - `l` (float or array-like floats) : The Galactic longitude coordinate(s) of the data, in degrees.

- `b` (float or array-like floats) : The Galactic latitude coordinate(s) of the data, in degrees.
- `r` (float or array-like floats) : The heliocentric distance(s) of the data.
- `normal` (array-like of floats) : The three-vector Cartesian coordinates for the normal vector of the plane.
- `point` (array-like of floats) : The three-vector Cartesian coordinates for the vector that suggests the new X-axis of the plane. Does not need to be orthogonal to the normal vector.

Returns:

- `Lam` (float or array-like floats) : The longitude coordinates of the points in the new planar coordinates.
- `Bet` (float or array-like floats) : The latitude coordinates of the points in the new planar coordinates.

- `mwahpy.coords.gal_to_lambet_galcentric(l, b, r, normal, point)`

  Takes in Galactic coordinates, the normal vector to a plane, and a point to suggest the new X-axis and returns longitude and latitude coordinates with the X-Y plane orthogonal to the normal vector and the Z-axis along the normal vector. This explicitly transforms l, b, r into Galactocentric Cartesian coordinates before transforming it into longitude and latitude, which `coords.gal_to_lambet` does not do. The longitude (Lambda) increases counter-clockwise as viewed looking on the plane from along the positive Z-axis, and the latitude (Beta) increases in the direction of the normal vector and is zero in the plane.

  Parameters:

  - `l` (float or array-like floats) : The Galactic longitude coordinate(s) of the data, in degrees.
  - `b` (float or array-like floats) : The Galactic latitude coordinate(s) of the data, in degrees.
  - `r` (float or array-like floats) : The heliocentric distance(s) of the data.
  - `normal` (array-like of floats) : The three-vector Cartesian coordinates for the normal vector of the plane.
  - `point` (array-like of floats) : The three-vector Cartesian coordinates for the vector that suggests the new X-axis of the plane. Does not need to be orthogonal to the normal vector.

  Returns:

  - `Lam` (float or array-like floats) : The longitude coordinates of the points in the new planar coordinates.
  - `Bet` (float or array-like floats) : The latitude coordinates of the points in the new planar coordinates.

- `mwahpy.coords.gal_to_plane(l, b, r, normal, point)`

  Takes in Galactic coordinates, the normal vector to a plane, and a point to suggest the new X-axis and returns shifted Cartesian coordinates with the X-Y plane orthogonal to the normal vector and the Z-axis along the normal vector.

  Parameters:

  - `l` (float or array-like floats) : The Galactic longitude coordinate(s) of the data.
  - `b` (float or array-like floats) : The Galactic latitude coordinate(s) of the data.
  - `r` (float or array-like floats) : The heliocentric distance(s) of the data.
  - `normal` (array-like of floats) : The three-vector Cartesian coordinates for the normal vector of the plane.
  - `point` (array-like of floats) : The three-vector Cartesian coordinates for the vector that suggests the new X-axis of the plane. Does not need to be orthogonal to the normal vector.

Returns:

- `x` (float or array-like floats) : The Cartesian X coordinates of the points in the new planar coordinates.
- `y` (float or array-like floats) : The Cartesian Y coordinates of the points in the new planar coordinates.
- `z` (float or array-like floats) : The Cartesian Z coordinates of the points in the new planar coordinates.

- `mwahpy.coords.gal_to_sgr(l, b, r)`

  Takes in Galactic coordinates and returns Sagittarius Tidal Stream longitude and latitude coordinates. See Majewski et al. (2003) for more info about the coordinate system.

  Parameters:

  - `l` (float or array-like floats) : The Galactic longitude coordinate(s) of the data.
  - `b` (float or array-like floats) : The Galactic latitude coordinate(s) of the data.
  - `r` (float or array-like floats) : The heliocentric distance(s) of the data.

  Returns:

  - `Lam` (float or array-like floats) : The longitude coordinates of the points in Sgr coordinates.
  - `Bet` (float or array-like floats) : The latitude coordinates of the points in Sgr coordinates.

- `mwahpy.coords.get_plane_normal(params)`

  Given normalized parameters `[a, b, c, d]` for a plane with the equation $ax + by + cz + d = 0$, returns a normal vector for that plane.

  Parameters:

  - `params` (array-like of floats) : The normalized parameters `[a, b, c, d]` for a plane with the equation $ax + by + cz + d = 0$.

  Returns:

  - `normal` (list of floats) : The normal vector to the given plane, as a three-vector in Galactocentric Cartesian coordinates.

- `mwahpy.coords.get_rvpm(ra, dec, dist, U, V, W)`

  This function uses an adaptation of the routine from Johnson & Soderblom (1987) to compute the radial velocity and proper motions of an object given its 3D position and velocities. If the velocities are input in the LSR frame (U, V, W), then the output will include solar reflex motion. If the velocities are input in the GSR (vx, vy, vz) frame, then the output will have the solar reflex motion removed.

  Parameters:

  - `ra` (float or `numpy` array of floats) : The right ascension(s) of the data (deg).
  - `dec` (float or `numpy` array of floats) : The declination(s) of the data (deg).
  - `dist` (float or `numpy` array of floats) : The heliocentric distance coordinate(s) of the data (kpc).
  - `U` (float or `numpy` array of floats) : The X component of the Cartesian velocity(ies) of the data (km/s).
  - `V` (float or `numpy` array of floats) : The Y component of the Cartesian velocity(ies) of the data (km/s).
  - `W` (float or `numpy` array of floats) : The Z component of the Cartesian velocity(ies) of the data (km/s).

Returns:

- `rv` (float or array-like floats) : The radial velocity(ies) of the data (km/s).
- `pmra` (float or array-like floats) : The right ascension (`cos(dec)`) proper motion(s) of the data (mas/yr).
- `pmde` (float or array-like floats) : The declination proper motion(s) of the data (mas/yr)

- `mwahpy.coords.get_uvw(ra, dec, dist, rv, pmra, pmde)`

  Given observational data, this function uses the routine from Johnson & Soderblom (1987) to compute each object's 3D Cartesian velocity in the local standard of rest (LSR) frame.

  Parameters:

  - `ra` (float or `numpy` array of floats) : The right ascension(s) of the data (deg).
  - `dec` (float or `numpy` array of floats) : The declination(s) of the data (deg).
  - `dist` (float or `numpy` array of floats) : The heliocentric distance(s) of the data (kpc).
  - `rv` (float or `numpy` array of floats) : The heliocentric (including solar reflex motion) radial velocity(ies) of the data (km/s).
  - `pmra` (float or `numpy` array of floats) : The proper motion(s) in right ascension of the data. NOTE: This should be already multiplied by a `cos(dec)` factor (mas/yr).
  - `pmdec` (float or `numpy` array of floats) : The proper motion(s) in declination of the data (mas/yr).

  Returns:

  - `U` (float or array-like floats) : The velocity in the X direction in the LSR frame (km/s).
  - `V` (float or array-like floats) : The velocity in the Y direction in the LSR frame (km/s).
  - `W` (float or array-like floats) : The velocity in the Z direction in the LSR frame (km/s).

- `mwahpy.coords.get_uvw_errors(ra, dec, dist, rv, pmra, pmde, err_pmra, err_pmdec, err_rv, err_dist)`

  Given observational data, this function uses the routine from Johnson & Soderblom (1987) to compute the uncertainty in each object's 3D Cartesian velocity in the local standard of rest (LSR) frame. This function assumes that the uncertainties in position on the sky are negligible, or at least much smaller than the uncertainties in the other parameters. Unlike `get_uvw()`, this function does not work for arrays.

  Parameters:

  - `ra` (float) : The right ascension of the data (deg).
  - `dec` (float) : The declination of the data (deg).
  - `dist` (float) : The heliocentric distance of the data (kpc).
  - `rv` (float) : The heliocentric (including solar reflex motion) radial velocity of the data (km/s).
  - `pmra` (float) : The proper motion in right ascension of the data. NOTE: This should be already multiplied by a `cos(dec)` factor (mas/yr).
  - `pmdec` (float) : The proper motion in declination of the data (mas/yr).
  - `err_pmra` (float) : The uncertainty in proper motion in right ascension of the data.
  - `err_pmdec` (float) : The uncertainty in proper motion in declination of the data (mas/yr).
  - `err_rv` (float) : The uncertainty in heliocentric (including solar reflex motion) radial velocity of the data (km/s).
  - `err_dist` (float) : The uncertainty in heliocentric distance of the data (kpc).

  Returns:

- `err_U` (float or array-like floats) : The uncertainty in the velocity in the X direction in the LSR frame (km/s).
- `err_V` (float or array-like floats) : The uncertainty in the velocity in the Y direction in the LSR frame (km/s).
- `err_W` (float or array-like floats) : The uncertainty in the velocity in the Z direction in the LSR frame (km/s).

- `mwahpy.coords.get_vxvyvz(ra, dec, dist, rv, pmra, pmdec)`

  Given observational data, this function uses the routine from Johnson & Soderblom (1987) to compute the 3D Galactic Cartesian coordinates for each object's velocity in the Galactic standard of rest ([V]GSR) frame (e.g. with solar reflex motions removed).

  Parameters:

  - `ra` (float or `numpy` array of floats) : The right ascension(s) of the data (deg).
  - `dec` (float or `numpy` array of floats) : The declination(s) of the data (deg).
  - `dist` (float or `numpy` array of floats) : The heliocentric distance(s) of the data (kpc).
  - `rv` (float or `numpy` array of floats) : The heliocentric (including solar reflex motion) radial velocity(ies) of the data (km/s).
  - `pmra` (float or `numpy` array of floats) : The proper motion(s) in right ascension of the data. NOTE: This should be already multiplied by a `cos(dec)` factor (mas/yr).
  - `pmdec` (float or `numpy` array of floats) : The proper motion(s) in declination of the data (mas/yr).

  Returns:

  - `vx` (float or array-like floats) : The velocity in the X direction in the GSR frame (km/s).
  - `vy` (float or array-like floats) : The velocity in the Y direction in the GSR frame (km/s).
  - `vz` (float or array-like floats) : The velocity in the Z direction in the GSR frame (km/s).

- `mwahpy.coords.plane_OLS(x, y, z, print_distances=False)`

  Takes in Galactocentric Cartesian coordinates of data and computes a best ordinary least-squares (OLS) fit for a plane through the data that minimizes the total distances between each point and the plane. Returns normalized parameters `[a, b, c]` for a plane with the equation $ax + by + cz = 0$. The plane is constrained to be required to go through the Galactic center.

  Parameters:

  - `x` (`numpy` array of floats) : The X Cartesian positions of the data.
  - `y` (`numpy` array of floats) : The Y Cartesian positions of the data.
  - `z` (`numpy` array of floats) : The Z Cartesian positions of the data.
  - `print_distances` (bool, optional: default=`False`) : If `True`, prints out the distances from each point from the fit plane.

  Returns:

  - `params` (list of floats) : The parameters `[a, b, c]` for the fit plane, which prescribe a plane with the equation $ax + by + cz = 0$.

- `mwahpy.coords.remove_sol_mot_from_pm(ra, dec, dist, pmra, pmdec)`

  Removes the solar reflex motion from the given proper motions. This assumes that `pmra` is `pmra*cos(dec)`.

  Parameters:

  - `ra` (float or `numpy` array of floats) : The right ascension(s) of the data (deg).
  - `dec` (float or `numpy` array of floats) : The declination(s) of the data (deg).

- `dist` (float or `numpy` array of floats) : The heliocentric distance coordinate(s) of the data (kpc).
- `pmra` (float or `numpy` array of floats) : The right ascension proper motion (multiplied by `cos(dec)`) (mas.yr).
- `pmdec` (float or `numpy` array of floats) : The declination proper motion (mas/yr).

Returns:

- `pmra` (float or array-like floats) : The right ascension (`cos(dec)`) proper motion(s) of the data with solar reflex motion removed (mas/yr).
- `pmde` (float or array-like floats) : The declination proper motion(s) of the data with solar reflex motion removed (mas/yr)

- `mwahpy.coords.rot_around_arb_axis(x, y, z, ux, uy, uz, theta)`

  Rotates a given point (x, y, z) about a given axis determined by $\hat{u} = \langle u_x, u_y, u_z \rangle$ by a given angle `theta`. The axis must pass through the origin, and points in the direction of $\hat{u}$. The data is rotated in a counterclockwise direction if viewed so that $\hat{u}$ is pointing towards you (out of the "page"). This function is not guaranteed to work for array-like values of `x`, `y`, `z`. To rotate systems into a frame determined by a pole and a longitudinal origin, see `mwahpy.coords.sky_to_pole()`, which does support array-like inputs.

  Parameters:

  - `x` (float) : The X coordinate of the data.
  - `y` (float) : The Y coordinate of the data.
  - `z` (float) : The Z coordinate of the data.
  - `ux` (float) : The X component of the axis vector $\hat{u}$. The components of $\hat{u}$ do not need to be normalized prior to input.
  - `uy` (float) : The Y component of the axis vector $\hat{u}$. The components of $\hat{u}$ do not need to be normalized prior to input.
  - `uz` (float) : The Z component of the axis vector $\hat{u}$. The components of $\hat{u}$ do not need to be normalized prior to input.
  - `theta` (float) : The angle in radians that the data is rotated around the axis.

  Returns:

  - `x` (float) : The new rotated X coordinate of the data.
  - `y` (float) : The new rotated Y coordinate of the data.
  - `z` (float) : The new rotated Z coordinate of the data.

- `mwahpy.coords.rv_to_vgsr(l, b, rv)`

  Takes in Galactic coordinates and a radial velocity returns the corresponding line-of-sight vleocity in the Galactic standard of rest. Supports array-like inputs.

  Parameters:

  - `l` (float or array-like floats) : The Galactic longitude coordinate(s) of the data, in degrees.
  - `b` (float or array-like floats) : The Galactic latitude coordinate(s) of the data, in degrees.
  - `rv` (float or array-like floats) : The heliocentric radial velocity(ies) of the data (km/s).

  Returns:

  - `vgsr` (float or array-like floats) : The line-of-sight velocity(ies) of the data (km/s).

- `mwahpy.coords.sgr_to_gal(Lam, Bet)`

  Takes in Sagittarius Tidal Stream longitude and latitude coordinates and returns Galactic coordinates. See Majewski et al. (2003) for more info about the coordinate system.

  Parameters:

  - `Lam` (float or array-like floats) : The longitude coordinates of the points in Sgr coordinates.
  - `Bet` (float or array-like floats) : The latitude coordinates of the points in Sgr coordinates.

  Returns:

  - `l` (float or array-like floats) : The Galactic longitude coordinate(s) of the data.
  - `b` (float or array-like floats) : The Galactic latitude coordinate(s) of the data.
  - `r` (float or array-like floats) : The heliocentric distance(s) of the data.

- `mwahpy.coords.sky_to_pole(sky1, sky2, pole, origin, wrap=False, rad=False)`

  Rotates on-sky coordinates into a new spherical coordinate system given by the new north pole position and a new origin position. Any spherical coordinate system will work as long as all inputs are in the same coordinates.

  Parameters:

  - `sky1` (float or array-like floats) : The longitude coordinates of the points.
  - `sky2` (float or array-like floats) : The latitude coordinates of the points.
  - `pole` (2-tuple-like of floats) : The longitude and latitude coordinates of the new pole, must be indexable.
  - `origin` (float or array-like floats) : The longitude and latitude coordinates of the new origin, must be indexable.
  - `wrap` (bool, optional: default=False) : If true, give the new longitude values as only positive values. Otherwse, the new longitude values span [-180, 180].
  - `rad` (bool, optional: default=False) : If true, inputs and outputs are in radians.

  Returns:

  - `Lam` (float or array-like floats) : The longitude coordinate(s) of the data in the rotated system.
  - `Bet` (float or array-like floats) : The latitude coordinate(s) of the data in the rotated system.

- `mwahpy.coords.spher_to_cart(phi, theta, r)`

  Takes in spherical coordinates and returns Cartesian coordinates. Supports array-like inputs.

  Parameters:

  - `phi` (float or array-like floats) : The azimuthal angle(s) of the data, in degrees.
  - `theta` (float or array-like floats) : The polar angle(s) of the data, in degrees.
  - `r` (float or array-like floats) : The spherical radius coordinate(s) of the data.

  Returns:

  - `x` (float or array-like floats) : The Cartesian X coordinate(s) of the data.
  - `y` (float or array-like floats) : The Cartesian Y coordinate(s) of the data.
  - `z` (float or array-like floats) : The Cartesian Z coordinate(s) of the data.

- `mwahpy.coords.wrap_long(long, rad=False)`

  Takes in a longitude or array-like collection of longitudes and returns it in the interval $[0, 360]$. If `rad` is set to `True`, then the longitude is instead returned in the interval $[0, 2\pi]$.

  Parameters:

– `long` (float or array-like floats) : The value(s) that will be cast into the interval. Should be in degrees by default, unless `rad` is set as `True`.

– `rad` (bool, optional) : If `True`, the input should be in radians and the output will be in radians. Otherwise, input should be in degrees and output will be in degrees.

Returns:

– `long` (float or array-like floats) : The value(s) that were passed in, cast into the interval $[0, 360]$ (or the equivalent in radians).

6.2 *orbit_fitter*

- `mwahpy.orbit_fitter.fit_orbit(l, b, b_err, d, d_err, vx=None, vy=None, vz=None, vgsr=None, vx_err=None, vy_err=None, vz_err=None, vgsr_err=None, max_it=100, bounds=[(0, 360), (-90, 90), (0, 100), (-500, 500), (-500, 500), (-500, 500)], t_len=None, **kwargs)`:

  Given observed particle data, will fit an orbit to that data by minimizing a chi-squared-like sum of squares parameter between the orbit and the data. Minimization is done using the `scipy.optimize.differential_evolution()` routine. Returns the best orbit fit parameters (a list of galactocentric longitude and latitude, heliocentric distance, and Galactocentric Cartesian velocities). Since differential evolution is not deterministic, a better idea of the best orbit fit may be obtained by running this routine several times on the dame data. Angles should be in degrees, distances in kpc, and velocities in km/s. The output is in these units.

  Parameters:

  – `l` (array-like of floats) : The Galactic longitudes of the observed data.

  – `b` (array-like of floats) : The Galactic latitudes of the observed data.

  – `b_err` (array-like of floats) : The errors in Galactic latitudes of the observed data.

  – `d` (array-like of floats) : The heliocentric distances of the observed data.

  – `d_err` (array-like of floats) : The errors in heliocentric distances of the observed data.

  – `vx` (optional, array-like of floats) : The galactocentric X velocities of the observed data. If not `None`, it is used for fitting.

  – `vy` (optional, array-like of floats) : The galactocentric Y velocities of the observed data. If not `None`, it is used for fitting.

  – `vz` (optional, array-like of floats) : The galactocentric Z velocities of the observed data. If not `None`, it is used for fitting.

  – `vgsr` (optional, array-like of floats) : The galactic standard of rest line-of-sight velocities of the observed data. If not `None`, it is used for fitting.

  – `vx_err` (optional, array-like of floats) : The errors in galactocentric X velocity of the observed data. If not `None`, it is used for fitting. Must be included if `vx` is included.

  – `vy_err` (optional, array-like of floats) : The errors in galactocentric Y velocity of the observed data. If not `None`, it is used for fitting. Must be included if `vy` is included.

  – `vz` (optional, array-like of floats) : The errors in galactocentric Z velocity of the observed data. If not `None`, it is used for fitting. Must be included if `vz` is included.

  – `vgsr_err` (optional, array-like of floats) : The errors in galactic standard of rest line-of-sight velocity of the observed data. If not `None`, it is used for fitting. Must be included if `vgsr` is included.

  – `max_it` (optional, int) : The maximum number of iterations that the differential evolution algorithm will run.

- `bounds` (optional, list of tuples of floats) : The bounds for the parameters (`l, b, d, vx, vy, vz`) of the data.
- `t_len` (optional, float) : The length that the test orbit will get evolved forwards and backwards in time, in Gyr.
- `**kwargs` (optional, keyword arguments) : Any additional keyword arguments will get passed to the `scipy.optimize.differential_evolution` routine.

Returns:

- `params` (list of floats) : The [`l, b, d, vx, vy, vz`] parameters for the calculated best orbit fit to the data.
- `x2` (float) : The goodness-of-fit of the calculated best orbit fit to the data.

- `mwahpy.orbit_fitter.make_orbit(params, int_ts=None)`

  Makes a corrected galpy orbit (vx is negated). Useful if one doesn't want to remember all the syntax for setting up an orbit with the proper right-handed velocities and the solar motion corrections to place the orbit in a Galactocentric frame.

  Parameters:

  - `params` (array-like of floats) : An array-like containing exactly six values: `l, b, d, vx, vy, vz`. These are the phase space parameters for the orbit in a right-handed heliocentric coordinate system, and should be in units of `[deg, deg, kpc, km/s, km/s, km/s]`.
  - `int_ts` (array-like of quantities) : The list of times that the orbit should be integrated forwards in time. Need to be quantities (preferably Gyr).

  Returns:

  - `o` (`galpy.orbit.Orbit` object) : The orbit (integrated for 0.5 Gyr forwards in time)

6.3 *orbit_fitter_gc*

- `mwahpy.orbit_fitter_gc.fit_orbit(l, b, b_err, d, d_err, vx=None, vy=None, vz=None, vgsr=None, vx_err=None, vy_err=None, vz_err=None, vgsr_err=None, max_it=100, bounds=[(0, 360), (-90, 90), (0, 100), (-500, 500), (-500, 500), (-500, 500)], t_len=None, **kwargs):`

  Given observed particle data, will fit an orbit to that data by minimizing a chi-squared-like sum of squares parameter between the orbit and the data. Minimization is done using the `scipy.optimize.differential_evolution()` routine. Returns the best orbit fit parameters (a list of galactocentric longitude and latitude, heliocentric distance, and Galactocentric Cartesian velocities). Since differential evolution is not deterministic, a better idea of the best orbit fit may be obtained by running this routine several times on the dame data. Angles should be in degrees, distances in kpc, and velocities in km/s. The output is in these units.

  The difference between this routine and the routine from `mwahpy.orbit_fitter` of the same name is that this routine places the data in a Galactocentric frame, computes a best ordinary least squares fit of a plane to the data (to approximate the orbital plane of the data in the Galaxy), and then fits using the new longitude and latitude of that rotated planar coordinate system.

  Parameters:

  - `l` (array-like of floats) : The Galactic longitudes of the observed data.
  - `b` (array-like of floats) : The Galactic latitudes of the observed data.
  - `b_err` (array-like of floats) : The errors in Galactic latitudes of the observed data.
  - `d` (array-like of floats) : The heliocentric distances of the observed data.
  - `d_err` (array-like of floats) : The errors in heliocentric distances of the observed data.

- `vx` (optional, array-like of floats) : The galactocentric X velocities of the observed data. If not `None`, it is used for fitting.
- `vy` (optional, array-like of floats) : The galactocentric Y velocities of the observed data. If not `None`, it is used for fitting.
- `vz` (optional, array-like of floats) : The galactocentric Z velocities of the observed data. If not `None`, it is used for fitting.
- `vgsr` (optional, array-like of floats) : The galactic standard of rest line-of-sight velocities of the observed data. If not `None`, it is used for fitting.
- `vx_err` (optional, array-like of floats) : The errors in galactocentric X velocity of the observed data. If not `None`, it is used for fitting. Must be included if `vx` is included.
- `vy_err` (optional, array-like of floats) : The errors in galactocentric Y velocity of the observed data. If not `None`, it is used for fitting. Must be included if `vy` is included.
- `vz` (optional, array-like of floats) : The errors in galactocentric Z velocity of the observed data. If not `None`, it is used for fitting. Must be included if `vz` is included.
- `vgsr_err` (optional, array-like of floats) : The errors in galactic standard of rest line-of-sight velocity of the observed data. If not `None`, it is used for fitting. Must be included if `vgsr` is included.
- `max_it` (optional, int) : The maximum number of iterations that the differential evolution algorithm will run.
- `bounds` (optional, list of tuples of floats) : The bounds for the parameters (`l, b, d, vx, vy, vz`) of the data.
- `t_len` (optional, float) : The length that the test orbit will get evolved forwards and backwards in time, in Gyr.
- `**kwargs` (optional, keyword arguments) : Any additional keyword arguments will get passed to the `scipy.optimize.differential_evolution` routine.

Returns:

- `params` (list of floats) : The [`l, b, d, vx, vy, vz`] parameters for the calculated best orbit fit to the data.
- `normal` (3-tuple of floats) : The components of the normal vector to the best-fit plane in Galactocentric Cartesian coordinates.
- `point` (3-tuple of floats) : The components of the point vector to the best-fit plane in Galactocentric Cartesian coordinates (Appriximately the new origin in the rotated frame).
- `x2` (float) : The goodness-of-fit of the calculated best orbit fit to the data.

6.4 *nbody*

These functions and methods are all related to the `Nbody` class, and are located in the `mwahpy/nbody.py` file. The `Nbody` class is meant to be used as a collection of `Timestep` instances, as in reading an entire folder of MilkyWay@home .out files that are the data for different times in the same simulation.

6.4.1 Initialization

An instance of the `Timestep` class is initialized by
`Nbody(ts={}, ts_scale=None)`.

- `ts` (dict of `Timesteps`) : The dictionary of `Timestep` instances that belong to the `Nbody` instance. The keys in the dictionary should be the given timestep (99, 2099, 1799, etc) in the .out filename used to read in each `Timestep`.
- `ts_scale` (float, optional: default=None) : If given, the keys of each `Timestep` in the `ts` dictionary are scaledby this value to get the actual time in time units instead of timesteps.

6.4.2 Methods

- `mwahpy.nbody.scale_times(l)`

  Replaces each key in the `Nbody.ts` dictionary with a value multiplied by `l`.

  Parameters:

  - `l` (float) : The time in physical units that is represented by a single timestep in the MW@h N-body simulation.

  Returns:

6.5 *orbit_fitter*

The subpackage `mwahpy.orbit_fitting` is still in development. Once the code has been refactored and brought to an acceptable level of quality, the documentation for this subpackage will be provided.

6.6 *orbit_fitter_gc*

The subpackage `mwahpy.orbit_fitting_gc` is still in development. Once the code has been refactored and brought to an acceptable level of quality, the documentation for this subpackage will be provided.

6.7 *output_handler*

The functions in this subpackage are meant for reading data in from files and printing data back out to files, in several different filetypes and in different scenarios.

6.7.1 Functions

- `mwahpy.output_handler.make_csv(t, f_name)`

  Writes out a .csv file given a `Timestep` instance.

  Parameters:

  - `t` (`Timestep` object) : The `Timestep` instance containing the data that you wish to write out.
  - `f_name` (str) : The path to the .csv file you wish to write to.

  Returns:

- `mwahpy.output_handler.make_nbody_input(t, f, recenter=True)`

  Writes out a .in file given a `Timestep` instance.

  Parameters:

  - `t` (`Timestep` object) : The `Timestep` instance containing the data that you wish to write out.
  - `f` (str) : The path to the .in file you wish to write to.
  - `recenter` (bool, optional: default=True) : If `True`, the data is recentered (via `Timestep.recenter()`) before writing out.

  Returns:

- `mwahpy.output_handler.read_folder(f)`

  Reads a folder full of .out files and returns an `Nbody` instance.

  Parameters:

  - `f` (str) : The path to the directory containing the files you wish to read in.

  Returns:

  - `n` (`Nbody` object) : An `Nbody` instance with its internal dictionary of `Timestep`s containing the data from the .out files in the folder.

- `mwahpy.output_handler.read_input(f)`

  Reads a .in file and returns a `Timestep` instance.

  Parameters:

  - `f` (str) : The path to the .in file you wish to read in.

  Returns:

  - `d` (`Timestep` object) : A `Timestep` instance containing the data from the .in file.

- `mwahpy.output_handler.read_output(f)`

  Reads a .out file and returns a `Timestep` instance.

  Parameters:

  - `f` (str) : The path to the .out file you wish to read in.

  Returns:

  - `d` (`Timestep` object) : A `Timestep` instance containing the data from the .out file.

- `mwahpy.output_handler.remove_header(f)`

  Removes the header from an open MilkyWay@home .out file and returns the center of mass and momentum information.

  Parameters:

  - `f` (open file) : An open .out file.

  Returns:

  - `comass` (list of floats) : The three-vector providing the center of mass of the system as per the .out file header.
  - `comom` (list of floats) : The three-vector providing the center of momentum of the system as per the .out file header.

6.8 *plot*

The subpackage `mwahpy.plot` is still in development. Some methods are accessible through `Timestep` methods, such as `Timestep.scatter()`. Once novel, useful code has been added that is not accessible in other ways, this section of the document will be fleshed out.

6.9 *timestep*

These functions and methods are all related to the `Timestep` class, and are located in the `mwahpy/timestep.py` file. These functions and methods make up the main core of *mwahpy* functionality, and rely on a lot of the other code in the package.

Additionally, a partial list of the many attributes of the `timestep` class are given below, and what they store is described in detail. Note that many attributes will return empty lists, empty arrays, `None`, or 0 initially if accessed without actually reading in or supplying data.

There are several other flags, functions, and attributes that are implemented in the `timestep.py` file that are not detailed below. These are not meant to be accessed by end users, and are explained by comments in the code.

6.9.1 Initialization

An instance of the `Timestep` class is initialized by

`Timestep(id_val=[], x=[], y=[], z=[], vx = [], vy = [], vz = [], mass=[], centerOfMass=[0, 0, 0], centerOfMomentum=[0, 0, 0], potential=None, time=None, nbody=None)`.

Note that all of the provided values between `id_val` and `mass` must be arrays of the same length, or else there will be errors.

- `typ` (list of ints, optional) : The types of each particle, where 0 represents a baryonic particle and 1 represents a dark matter particle (in order).
- `id_val` (list of ints, optional) : The id values of each particle (in order).
- `x` (list of floats, optional) : The Galactocentric X positions of each particle in units of kpc (in order).
- `y` (list of floats, optional) : The Galactocentric Y positions of each particle in units of kpc (in order).
- `z` (list of floats, optional) : The Galactocentric Z positions of each particle in units of kpc (in order).
- `vx` (list of floats, optional) : The Galactic standard of rest velocities of each particle in the X direction in units of km/s (in order).
- `vy` (list of floats, optional) : The Galactic standard of rest velocities of each particle in the Y direction in units of km/s (in order).
- `vz` (list of floats, optional) : The Galactic standard of rest velocities of each particle in the Z direction in units of km/s (in order).
- `mass` (list of floats, optional) : The masses of each particle in units of MilkyWay@home structure masses (in order).
- `center_of_mass` (list of floats, optional) : The 3D location of the center of mass of the system (all particles in the `Timestep` instance) in units of kpc. The first element of the list is the Galactocentric X position of the center of mass, while the second and third elements are the Y and Z positions, respectively.
- `center_of_momentum` (list of floats, optional) : The 3D location of the center of momentum of the system (all particles in the `Timestep` instance) in units of km/s. The first element of the list is the Galactocentric vx position of the center of mass, while the second and third elements are the vy and vz positions, respectively.
- `potential` (`galpy` potential object or list of `galpy` potential objects, optional) : Can be `None`. If not `None`, the energy of each particle will be calculated with this potential instead of the default *mwahpy* potential.
- `time` (float, optional) : The time in Gyr after the beginning of the simulation.
- `nbody` (`Nbody` object, optional) : The parent `Nbody` instance for this `Timestep`.

6.9.2 Attributes

- `mwahpy.timestep.b`

  (`numpy` array of floats) : The Galactic latitude of each particle in degrees, in order. Calculated value.

- `mwahpy.timestep.centerOfMass`

  (list of floats) : The 3D location of the center of mass of the system (all particles in the `Timestep` instance) in units of kpc. The first element of the list is the Galactocentric X position of the center of mass, while the second and third elements are the Y and Z positions, respectively.

- `mwahpy.timestep.centerOfMomentum`

  (list of floats) : The 3D location of the center of momentum of the system (all particles in the `Timestep` instance) in units of km/s. The first element of the list is the Galactocentric vx position of the center of mass, while the second and third elements are the vy and vz positions, respectively.

- `mwahpy.timestep.dec`

  (`numpy` array of floats) : The declination of each particle in degrees, in order. Calculated value. Computed using `astropy`.

- `mwahpy.timestep.dist`

  (`numpy` array of floats) : The heliocentric spherical radius of each particle in units of kpc, in order. Equal to $\sqrt{(X+8)^2 + Y^2 + Z^2}$. Calculated value.

- `mwahpy.timestep.distFromCOM`

  (`numpy` array of floats) : The distance of each particle from the center of mass of the `Timestep` instance, in order. Calculated value.

- `mwahpy.timestep.energy`

  (`numpy` array of floats) : The total energy per unit mass of each particle based on the specified potential in units of km^2/s^2 (in order). If no potential is specified, this is calculated using the default *mwahpy* potential. Calculated value as part of `calc_energy()`.

- `mwahpy.timestep.id`

  (`numpy` array of ints) : The id of each particle (in order).

- `mwahpy.timestep.index_list`

  (list of strs) : A list of values that are currently calculated for the `Timestep`. This is automatically updated as new values are calculated. Examples of list members are 'x', 'mass', 'vlos', etc. This is used by *mwahpy* to determine what values need to be calculated and/or iterated over, and therefore it is strongly recommended to not alter this attribute manually.

- `mwahpy.timestep.KE`

  (`numpy` array of floats) : The kinetic energy per unit mass of each particle in units of km^2/s^2 (in order). Calculated value as part of `calc_energy()`.

- `mwahpy.timestep.l`

  (`numpy` array of floats) : The Galactic longitude of each particle in degrees, in order. Calculated value.

- `mwahpy.timestep.lperp`

  (`numpy` array of floats) : The magnitude of the projection of the angular momentum of each particle onto the $X - Y$ plane, in order. Has units of kpc·km/s. Equal to $\sqrt{L_x^2 + L_y^2}$. Calculated value.

- `mwahpy.timestep.ltot`

  (`numpy` array of floats) : The magnitude of the total angular momentum of each particle in units of kpc·km/s, in order. Calculated value.

- `mwahpy.timestep.lx`

  (`numpy` array of floats) : The X component of the angular momentum of each particle about the origin in units of kpc·km/s, in order. Calculated value.

- `mwahpy.timestep.ly`

  (`numpy` array of floats) : The Y component of the angular momentum of each particle about the origin in units of kpc·km/s, in order. Calculated value.

- `mwahpy.timestep.lz`

  (`numpy` array of floats) : The Z component of the angular momentum of each particle about the origin in units of kpc·km/s, in order. Calculated value.

- `mwahpy.timestep.mass`

  (`numpy` array of floats) : The mass of each particle in units of MilkyWay@home structure masses (in order).

- `mwahpy.timestep.msol`

  (`numpy` array of floats) : The mass of each particle in units of solar masses (in order). Calculated value.

- `mwahpy.timestep.nbody`

  (`nbody` object) : If the `Timestep` instance is part of an `nbody` instance, i.e. if you read in the `Timestep` with `mwahpy.output_handler.read_folder`, then that information is saved here.

- `mwahpy.timestep.PE`

  (`numpy` array of floats) : The potential energy per unit mass of each particle based on the specified potential in units of km^2/s^2 (in order). If no potential is specified, this is calculated using the default *mwahpy* potential. Calculated value as part of `calc_energy()`.

- `mwahpy.timestep.phi`

  (`numpy` array of floats) : Galactocentric spherical aximuthal angle in degrees, where 0 lies along the Galactic Cartesian X axis and increases in the same direction as Galactic longtitude l. Calculated value.

- `mwahpy.timestep.pmdec`

  (`numpy` array of floats) : The proper motion of each particle in the direction of declination in units of mas/yr (in order). Calculated value as part of `calc_rvpm()`.

- `mwahpy.timestep.pmra`

  (`numpy` array of floats) : The proper motion of each particle in the direction of right ascension in units of mas/yr (in order). Multiplied by $\cos\delta$. Calculated value as part of `calc_rvpm()`.

- `mwahpy.timestep.pmtot`

  (`numpy` array of floats) : The magnitude of the total proper motion of each particle in units of mas/yr (in order). Calculated value as part of `calc_rvpm()`.

- `mwahpy.timestep.potential`

  (`galpy` potential object or list of `galpy` potential objects) : If not `None`, the energy of each particle will be calculated with this potential. If `None`, the *mwahpy* default potential will be used to calculate the particle energies.

- `mwahpy.timestep.r`

  (`numpy` array of floats) : The Galactocentric spherical radius of each particle in units of kpc, in order. Equal to $\sqrt{X^2 + Y^2 + Z^2}$. Calculated value.

- `mwahpy.timestep.R`

  (`numpy` array of floats) : The Galactocentric cylindrical radius of each particle in units of kpc, in order. Equal to $\sqrt{X^2 + Y^2}$. Calculated value.

- `mwahpy.timestep.ra`

  (`numpy` array of floats) : The right ascension of each particle in degrees, in order. Calculated value. Computed using `astropy`.

- `mwahpy.timestep.theta`

  (`numpy` array of floats) : Galactocentric spherical inclination angle in degrees, where 0 lies in the Galactic Cartesian X-Y plane and increases/decreases in the same directions as Galactic longtitude b. Calculated value.

- `mwahpy.timestep.time`

  (float) : The time in Gyr for this `Timestep` instance. This will only be initialized if the `Timestep` was read in as part of `mwahpy.output_handler.read_folder()` or if manually specified. Otherwise, will return `None` by default.

- `mwahpy.timestep.typ`

  (`numpy` array of ints) : The types of each particle, where 0 corresponds to a baryonic particle and 1 corresponds to a dark matter particle, in order.

- `mwahpy.timestep.vgsr`

  (`numpy` array of floats) : The line-of-sight velocity of each particle in the Galactic standard of rest in units of km/s, in order.

- `mwahpy.timestep.vlos`

  (`numpy` array of floats) : The heliocentric line-of-sight velocity of each particle in units of km/s, in order.

- `mwahpy.timestep.vrad`

  (`numpy` array of floats) : The magnitude of the Galactic standard of rest line-of-sight velocity of each particle as seen from the origin. Has units of km/s. Calculated value.

- `mwahpy.timestep.vrot`

  (`numpy` array of floats) : The magnitude of the Galactic standard of rest rotational velocity (projected onto the $X - Y$ plane) of each particle as seen from the origin. Has units of km/s. Calculated value.

- `mwahpy.timestep.vtan`

  (`numpy` array of floats) : The heliocentric, Galactic standard of rest tangential velocity of each particle in units of km/s (in order). Calculated value as part of `calc_rvpm()`.

- `mwahpy.timestep.vx`

  (`numpy` array of floats) : The Galactic standard of rest velocity in the X direction of each particle in units of km/s (in order).

- `mwahpy.timestep.vy`

  (`numpy` array of floats) : The Galactic standard of rest velocity in the Y direction of each particle in units of km/s (in order).

- `mwahpy.timestep.vz`

  (`numpy` array of floats) : The Galactic standard of rest velocity in the Z direction of each particle in units of km/s (in order).

- `mwahpy.timestep.x`

  (`numpy` array of floats) : The Galactocentric X position of each particle in units of kpc (in order).

- `mwahpy.timestep.y`

  (`numpy` array of floats) : The Galactocentric Y position of each particle in units of kpc (in order).

- `mwahpy.timestep.z`

  (`numpy` array of floats) : The Galactocentric Z position of each particle in units of kpc (in order).

6.9.3 Methods

- `mwahpy.timestep.append_timestep(t)`

  Combines two `Timestep` objects by appending the particles from the provided `Timestep` onto this `Timestep`.

  Parameters:

  - `t` (`Timestep` object) : The `Timestep` object that you wish to combine with this one.

  Returns:

- `mwahpy.timestep.append_point(t, n=0, id=None)`

  Adds a specific particle from the provided `Timestep` to this `Timestep`. What particle you wish to append can be specified by the `id` of the particle, or what position it is located at in the specified `Timestep` object.

  WARNING: This function is extremely time intensive if used repeatedly due to the overhead associated with `numpy` arrays. One should always try to use combinations of cuts and `mwahpy.timestep.append_timestep()` to append multiple particles whenever possible.

  Parameters:

  - `t` (`Timestep` object) : The `Timestep` object containing the particle that you wish to add to this `Timestep`.
  - `n` (int, optional) : The location of the particle in `t` that you wish to add to this `Timestep`. Ignored if `id` is not `None`.
  - `id` (int, optional) : If not `None`, this function locates the particle with `id` equal to this value and adds it to this `Timestep`.

  Returns:

- `mwahpy.timestep.copy()`

  Creates a deep copy of the `Timestep` instance. No arrays or lists in the copy are aliased to the original respective objects. Will not assign the corresponding "pointer" in a parent `Nbody` instance to the new copy.

  Parameters:

  Returns:

  - `out` (Timestep) : A deep copy of the `Timestep` instance.

- `mwahpy.timestep.cut_first_n(n)`

  Removes the first n particles from the `Timestep` instance, based on the particle order.

  Parameters:

  - `n` (int) : The number of particles to remove from the front of the `Timestep`.

  Returns:

- `mwahpy.timestep.cut_last_n(n)`

  Removes the last n particles from the `Timestep` instance, based on the particle order.

  Parameters:

  - `n` (int) : The number of particles to remove from the back of the `Timestep`.

  Returns:

- `mwahpy.timestep.hist2d(x, y, show=False, **kwargs)`

  Creates and optionally shows a 2D histogram plot of the particle data in the `Timestep`. See `mwahpy.plot.hist2d()` for a more in-depth explanation of the plotting routine.

  Parameters:

  - `x` (str) : The values that you want to plot on the horizontal axis. For example, if you want to plot `vlos` on the horizontal axis, you would input `'vlos'`.
  - `y` (str) : The values that you want to plot on the vertical axis (see `x` above).
  - `show` (optional, default = `False`) : If `True`, show the output plot. Otherwise, it is kept as the active figure to draw on/over.
  - `**kwargs` (optional) : Keyword arguments that you want to pass to `mwahpy.plot.hist2d()` or `matplotlib.pyplot.hist2d`. See the documentation for those functions for more information about what keyword arguments are supported.

  Returns:

- `mwahpy.timestep.len()`

  Returns the length of the timestep's values arrays. This is only calculated from the `timestep.id` array, so if that is out of sync with the other arrays for some reason this will not return an accurate value. Alternatively, one can use `len(Timestep)` for the same result.

  Parameters:

  Returns:

  - `len` (int) : The length of the timestep's values arrays.

- `mwahpy.timestep.only_baryons()`

  Removes all dark matter (`typ == 1`) particles from the `Timestep`.

  Parameters:

  Returns:

- `mwahpy.timestep.only_dark_matter()`

  Removes all baryonic (`typ == 0`) particles from the `Timestep`.

  Parameters:

  Returns:

- `mwahpy.timestep.print_particle(n, dec=8)`

  Prints information about all values corresponding to a single particle in the console.

  Parameters:

  - `n` (int) : The location of the particle in this `Timestep` that you want to print the information for.
  - `dec` (int, optional) : The number of digits after the decimal point to display for each value.

  Returns:

- `mwahpy.timestep.rand_sample(n)`

  Randomly samples the `Timestep` down to exactly n particles selected at random from the `Timestep`. Uses the reservoir theorem to sample the data.

  Parameters:

  - `n` (int) : The number of particles to include in the random sample.

  Returns:

- `mwahpy.timestep.recenter()`

  Shifts the positions and velocities of each particle by the same amount in order to make the center of mass and center of momentum both be located at the origin. This function does not change any physics, it just changes the frame in which the `Timestep` resides.

  Parameters:

  Returns:

- `mwahpy.timestep.reset_ids()`

  Relabels the `id` of each particle, in order, starting from 0 and incrementing by 1 each particle. Retains the current order of the particles.

  Parameters:

  Returns:

- `mwahpy.timestep.rot_around_z_axis(theta)`

  Rotates the positions and velocities of each particle by `theta` radians counterclockwise (as viewed "looking down" from positive z).

  Parameters:

  - `theta` (float) : The angle to rotate the data in radians.

  Returns:

- `mwahpy.timestep.scatter(x, y, **kwargs)`

  Creates and optionally shows a scatter plot of the particle data in the `Timestep`. See `mwahpy.plot.scatter()` for a more in-depth explanation of the plotting routine.

  Parameters:

  - `x` (str) : The values that you want to plot on the horizontal axis. For example, if you want to plot `vlos` on the horizontal axis, you would input `'vlos'`.
  - `y` (str) : The values that you want to plot on the vertical axis (see `x` above).
  - `**kwargs` (optional) : Keyword arguments that you want to pass to `mwahpy.plot.scatter()` or `matplotlib.pyplot.scatter`. See the documentation for those functions for more information about what keyword arguments are supported.

  Returns:

- `mwahpy.timestep.split(n)`

  Splits the `Timestep` into two new `Timestep` instances at the nth particle. The first output `Timestep` will contain the first n objects of the original `Timestep`, and the second output `Timestep` will contain the rest of the particles. Does not alter the original `Timestep`.

  Parameters:

  - `n` (int) : The location at which to split the `Timestep`.

  Returns:

  - `Timestep1` (`Timestep` object) : A `Timestep` instance containing the first n particles from the original `Timestep`.
  - `Timestep2` (`Timestep` object) : A `Timestep` instance containing the rest of the particles from the original `Timestep` that are not included in `Timestep1`.

- `mwahpy.timestep.split_at_id_wrap()`

  Splits the `Timestep` at each location where the `id` wraps back to 0. This is useful for simulations with multiple components. See `mwahpy.timestep.split()` for more information about splitting `Timestep` objects. The original `Timestep` object is not altered.

  WARNING: Under some conditions, MilkyWay@home can begin indexing particles in a MilkyWay@home generated dwarf (or dwarf component[s]) starting with 1, instead of 0. In this case, the particles will have to be manually split. This is a known bug in MilkyWay@home.

  Parameters:

  Returns:

  - `outlist` (list of `Timestep` objects) : A list containing a new `Timestep` instance made up of each component in the original `Timestep` object (in this case a component is defined by the particles in between two `id`s that are 0 in a `Timestep`). Each particle from the original `Timestep` belongs to exactly 1 component.

- `mwahpy.timestep.subsample(n, offset=0)`

  Uniformly samples the `Timestep` by taking every nth particle starting from the `offset`.

  Parameters:

  - `n` (int) : The number of particles to skip between each sample particle.
  - `offset` (int, optional) : The number of particles to initially skip.

  Returns:

- `mwahpy.timestep.subset_rect(axes, bounds)`

  Takes a multidimensional rectangular cut on the `Timestep`. The number of cuts made and their extents are defined by `axes` and `bounds`. For each axis specified in `axes`, all particles with values outside the respective `bounds` are removed from the `Timestep`. Axes may be reused (if, for example, one wanted to remove all particles within some rectangular boundary instead of all particles outside it).

  If you wish to only declare a minimum or maximum value for your data, then set the other bound as `None` (see `bounds`).

  Parameters:

  - `axes` (list of strs) : Each axis that will be cut on. For example, if you want to make a cut in proper motion, a cut in line-of-sight velocity, and a cut in distance, you would write `['pmra', 'vlos', 'dist']`. Order does not matter, but the order of the boundaries in `bounds` and the axes in `axes` must be the same.
  - `bounds` (list of tuples of floats) : The boundaries for each cut. For the example above in `axes`, if you wanted to include particles only with proper motions between -2 mas/yr and 2 mas/yr, line-of-sight velocity above 0 km/s, and distance less than 10 kpc, you would write `[(-2, 2), (0, None), (None, 10)]`. The order of the boundaries must be the same as the order of the axes specified in `axes`.

  Returns:

- `mwahpy.timestep.subset_circ(axes, rads, centers)`

  Takes a multidimensional "circular" cut on the `Timestep`. The number of cuts made and their extents are defined by `axes`, `rads`, and `centers`. For each axis specified in `axes`, all particles with values outside the specified region are removed from the `Timestep`. Axes may be reused.

  For example, if one wanted all particles with position within 2 kpc of the point $(X, Y) = (10, 15)$ kpc, one could make `axes` equal to `['x', 'y']`, make `rads` equal to `[2, 2]`, and make centers equal to `[10, 15]`.

  Parameters:

- `axes` (list of strs) : Each axis that will be cut on. Order does not matter, but the order of the boundaries in `rads` and `centers` and the order of the axes given in `axes` must be the same.
- `rads` (list of floats) : The radii of each cut. The order of the radii must be the same as the order of the axes specified in `axes`.
- `centers` (list of floats) : The centers of each cut. The order of the centers must be the same as the order of the axes specified in `axes`.

Returns:

- `mwahpy.timestep.take(indices)`

  Cuts the `Timestep` to only include the particles with indices equal to those of `indices`. WARNING: If the first particle in the `Timestep` has an `id` of 1, then this will be off by 1 from the particle `id`. In that case, `Timestep.take_by_id()` may have the desired behavior instead of this function.

  Parameters:

  - `indices` (list of ints) : The list of indices. Only particles with indices equal to a value in this list will be in the cut `Timestep`.

  Returns:

- `mwahpy.timestep.take_by_id(ids)`

  Cuts the `Timestep` to only include the particles with the specified values of `id`.

  Parameters:

  - `ids` (list of ints) : The list of `id`s. Only particles with `id` equal to a value in this list will be in the cut `Timestep`.

  Returns:

- `mwahpy.timestep.update(force=False)`

  Updates the `center_of_mass` and `center_of_momentum` attributes of the `Timestep` instance based on the current positions and velocities of all particles in the `Timestep`. Also updates all calculated values. Can be time intensive depending on how much needs to be updated, and whether the positions/velocities of the particles were actually changed (this is tracked in the `Timestep` implementation, and items are only updated if necessary).

  If `force` is set to `True`, then a complete update of all attributes of the `Timestep` will happen even if no positions or velocities have changed since the last update or initialization of the `Timestep`.

  Parameters:

  - `force` (bool, optional) : If true, force a complete update of the `Timestep` regardless of what has changed since the last update.

  Returns:

6.9.4 Functions

- `mwahpy.timestep.get_self_energies(t)`

  Computes the energy of each particle based only on the self-gravity of the particles in the `Timestep`, as well as their velocities with respect to their own centers of mass and momentum.

  Parameters:

  - `t` (Timestep) : The timestep that you are calculating the energies of.

  Returns:

– `energies` (`numpy` array of floats) : The calculated energies for all of the particles in the `Timestep`.

- `mwahpy.timestep.find_progenitor(t, ran=100.)`

  Identifies where the progenitor of a stream is. More accurately, it computes where the highest concentration of baryons is in on-sky position in the `Timestep` instance.

  Parameters:

  – `t` (Timestep) : The timestep that contains the data you are analyzing.

  – `ran` (float, optional: default=100.) : The maximum (heliocentric) distance range to search, in kpc.

  Returns:

  – `pos` (list of floats) : The Galactic Cartesian coordinate positions of the progenitor. If none was found, returns `[0, 0, 0]`.